UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

Doble Grado en Ingeniería Informática y Matemáticas

TRABAJO FIN DE GRADO

Aplicación para la estimación de ritmo cardiaco desde una cámara web

Javier Montalvo Rodrigo
Tutor: José M. Martínez Sánchez

Junio 2020

Aplicación para la estimación de ritmo cardiaco desde una cámara web

AUTOR: Javier Montalvo Rodrigo
TUTOR: José María Martínez Sánchez

Video Processing and Understanding Lab
Dpto. Tecnología Electrónica y de las Comunicaciones
Escuela Politécnica Superior
Universidad Autónoma de Madrid
Junio de 2020

Resumen

Medir la frecuencia cardiaca nos permite obtener información no solo sobre el estado y la salud cardiovascular de un sujeto, sino también sobre posibles enfermedades y problemas subyacentes, como un fallo renal o estrés entre otros. La frecuencia cardiaca se puede obtener de distintas formas, pero este trabajo se centra en la estimación de la frecuencia cardiaca a partir de videos.

El objetivo de este Trabajo Fin de Grado es ampliar y mejorar una aplicación ya existente que para que pueda ser utilizada como una herramienta para el desarrollo de algoritmos que estimen la frecuencia cardiaca a partir de secuencias de video.

La finalidad es tener una aplicación que sea capaz de ejecutar algoritmos directamente sobre videos previamente grabados y almacenados en un fichero, y sobre video capturado en directo, la cual debe disponer de varios algoritmos y estar preparada para que se puedan añadir nuevos algoritmos de manera modular.

La primera parte del trabajo ha consistido en migrar unos algoritmos basados en fotopletismografía y magnificación Euleriana, previamente desarrollados en Matlab, a C++ y C# haciendo uso de la librería OpenCV, de código abierto. También se ha mejorado el algoritmo ya existente en la propia aplicación para hacerlo más preciso y eficiente.

La segunda parte del trabajo se ha centrado en el desarrollo de un nuevo módulo para la aplicación que sea capaz de ejecutar algoritmos para estimar la frecuencia cardiaca sobre un video en directo obtenido a partir de una cámara web.

Por último, se han integrado los algoritmos sobre los que se ha trabajado en la primera parte para que puedan ser utilizados desde la aplicación y se ha comprobado el correcto funcionamiento de éstos, tanto en el modo ya existente que estimaba a partir de ficheros de video como en el modo nuevo a partir de un video en directo,

Palabras clave

Frecuencia Cardiaca, Fotopletismografía, Magnificación Euleriana, Análisis de Secuencias de Video, Cámara Web, OpenCV, visión por ordenador.

Abstract

Measuring heart rate allows us to obtain information not only about the cardiovascular status and health of a person, but also of possible underlying diseases or illnesses like renal failure or anxiety, among others. The heart rate can be obtained using different techniques and devices, but this thesis focuses on obtaining heart rate analyzing video sequences.

This Bachelor Thesis objective is improving and extending a preexistent application so it can be used as a tool for the development of new algorithms to estimate heart rate from videos.

The purpose is to have an application that can be used to record videos, execute different algorithms to estimate heart rate from these pre-recorded videos and from live videos that will be captured using a webcam. It will also support a more modular approach to the inclusion of new algorithms.

The first part of this work consists on the study and understanding of some algorithms based on Eulerian magnification and photoplethysmography, that were previously developed in Matlab, to migrate these to C++ and C# using the open source library OpenCV. Another algorithm that was already implemented in the preexistent application has been improved to work faster and more accurately.

The second part focuses on the development of a new module for the preexistent application that will be able to execute algorithms to estimate heart rate from a live feed obtained via a webcam.

At last, the previously migrated algorithms have been integrated in the application so they can be used for estimation of heart rate from both live feed mode and over pre-recorded files mode, and its correct functioning and performance have been tested.

Keywords

Heart Rate, Photoplethysmography, Eulerian Magnification, Video Sequences Analysis, webcam, OpenCV, Computer Vision.

Agradecimientos

Quiero comenzar agradeciendo a mis padres el brindarme la oportunidad de realizar estos estudios, y ayudarme en todo lo posible con ellos. A mis hermanas por estar ahí cuando algo no me salía como debía y necesitaba alguien con quien hablar.

A mi tutor, Chema, por confiar en mi para desarrollar este trabajo, y por haberme ayudado cuando lo he necesitado, y por la dedicación para ayudarme a mejorar durante el desarrollo de este trabajo.

A Jess por hacer mi vida más entretenida y alegrarme cuando algo no salía bien, y a Carlos por ayudarme siempre que necesitaba una crítica constructiva y honesta.

INDICE DE CONTENIDOS

INDICE DE FIGURAS

INDICE DE TABLAS

1 Introducción

1.1 Motivación

Medir la frecuencia cardiaca y su variabilidad nos permite detectar enfermedades como problemas cardiovasculares, fallos renales, diabetes, cirrosis de hígado o sepsis entre otros.

Las técnicas tradicionales para medir la frecuencia cardiaca requieren por lo general contacto físico, pero hay personas para las cuales esto puede no ser una posibilidad. Las personas mayores, los bebés prematuros, o algunas personas que presentan afecciones cutáneas pueden recibir daños en la piel al aplicarles electrodos o un oxímetro. También resulta interesante poder medir la frecuencia cardiaca en remoto de forma no intrusiva para poder realizar un control de salud mientras se duerme, se ve el televisor o se realizan actividades deportivas. Por estos motivos, desde principios de siglo se está investigando en métodos para obtener la frecuencia cardiaca a distancia.

Obtener la frecuencia cardiaca analizando secuencias de vídeo es el método en el que se centra este trabajo. La principal ventaja que ofrece este método respecto a otros es que cualquier cámara podría utilizarse para realizar una monitorización de la frecuencia cardiaca, desde cámaras de *smartphones*, de portátiles, de vigilancia, de equipos deportivos, consolas, … Todas estas cámaras podrían utilizarse para monitorizar la frecuencia cardiaca de una (o varias) personas.

En concreto, este trabajo se centra en seguir trabajando en la línea en la que se lleva trabajando en el VPULab, donde se cuenta con una serie de algoritmos y una aplicación que se centran en obtener la frecuencia cardiaca a partir de vídeo. El objetivo es mejorar esta aplicación, permitiendo obtener la frecuencia cardiaca en tiempo real a partir de una *webcam*, integrar más algoritmos y mejorar la funcionalidad actual de la propia aplicación.

1.2 Objetivos

Los objetivos para este trabajo son:

- Entender y migrar los algoritmos del VPULab desde Matlab para ser compatibles con OpenCV.
- Ampliar la aplicación del VPULab para que pueda funcionar con un video en tiempo real, mediante una *webcam*.
- Incorporar los algoritmos ya migrados a OpenCV a la aplicación para poder evaluar y contrastar su funcionamiento de manera más directa e intuitiva.

1.3 Organización de la memoria

La memoria consta de los siguientes capítulos:

- Capítulo 1 – Introducción. Motivación y objetivos del trabajo.
- Capítulo 2 – Estado del Arte. Métodos para obtener la frecuencia cardiaca y estado actual de la obtención de dicha frecuencia mediante análisis de video.

- Capítulo 3 – Diseño. Diseño de los Algoritmos para ser implementados usando OpenCV. Diseño del módulo para análisis de video en tiempo real en la aplicación y rediseñar el módulo de Algoritmos para facilitar la integración de nuevos algoritmos.
- Capítulo 4 – Desarrollo. Implementación de los Algoritmos en C++. Implementación del módulo en tiempo real usando una *webcam* e implementación de los cambios en el módulo de Algoritmos. Integración de los algoritmos implementados en la aplicación
- Capítulo 5 – Pruebas y resultados. Ejemplo del estado de la aplicación final. Pruebas realizadas sobre diferentes vídeos con los diferentes algoritmos. Evaluación de los cambios y mejoras realizados en los algoritmos integrados. Limitaciones encontradas.
- Capítulo 6 – Conclusiones y trabajo futuro.

2 Estado del arte

2.1 Funcionamiento del corazón

El corazón es un órgano muscular cuya labor es bombear sangre por el sistema circulatorio. Estos bombeos de sangre son autocontrolados, y se realizan de manera cíclica y rítmica.

En cada ciclo o latido podemos distinguir distintas fases. La primera fase, conocida como diástole, el corazón se llena de sangre, la cual es bombeada en la segunda fase, llamada sístole, a las diferentes arterias, que la distribuyen por el resto del cuerpo. Estos cambios de fase producen diferentes efectos en el cuerpo como, por ejemplo, cambios en la coloración de la piel (siendo esta ligeramente más pálida en la fase de diástole que en la sístole) o el movimiento de la cabeza al recibir sangre.

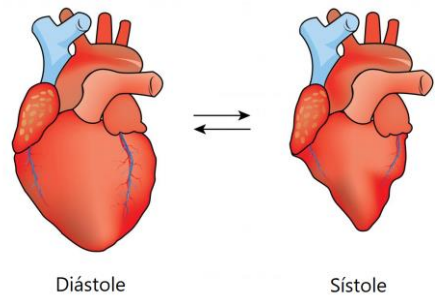

Figura 2-1. Fases del latido [1]

La cantidad de estos ciclos que suceden en un periodo de tiempo es lo que conocemos como frecuencia cardiaca. La medición de dicha frecuencia tiene diversas utilidades en salud, desde la detección de posibles problemas cardiacos, a otros problemas como sepsis o cirrosis de hígado.

Esta frecuencia se puede medir de diferentes maneras, siendo los electrocardiogramas (o ECG) los más extendidos. Este método consiste en medir el voltaje a lo largo del tiempo de la actividad eléctrica del corazón, mediante electrodos colocados en la piel, que detectan los cambios producidos al despolarizarse el corazón (sístole) y repolarizarse (diástole) en cada ciclo cardiaco.

Existen otros métodos para medir la frecuencia cardiaca, como la vibrocardiografía (VCG), que se basa en medir las vibraciones de la superficie de la piel causadas por el movimiento de las paredes arteriales.

Una de las limitaciones que nos ofrecen los métodos como los ECG y la vibrocardiografia, es que necesitan que haya un contacto físico con la persona a la que se desea medir la frecuencia cardiaca, y esto no siempre es una posibilidad. Las personas de avanzada edad, neonatos, o gente con enfermedades como la epidermólisis ampollar, pueden recibir daños cutáneos al utilizar un sensor tradicional, por lo que las alternativas para medir el pulso no invasivas se convierten en una necesidad en estos casos.

Algunas de las técnicas utilizadas para medir frecuencia cardiaca de manera no invasiva se basan en electromagnetismo, como microondas [2] o Wi-Fi [3], otras utilizan láseres para realizar vibrocardiografía a distancia [4], y otras se basan en algoritmos aplicados sobre imágenes de video para calcular la frecuencia cardiaca. Estas últimas son en las que se centra este trabajo.

2.2 Métodos para medir frecuencia cardíaca mediante análisis de video

Hay algunos algoritmos preparados para calcular la frecuencia cardiaca a partir de imágenes obtenidas con equipos más especializado, como cámaras térmicas [5] o cámaras de infrarrojos[6], pero en los que nos vamos a centrar son aquellos diseñados para funcionar con una cámara estándar (salida de un vídeo en color RGB), como las que podemos encontrar en teléfonos móviles, ordenadores y otros dispositivos.

2.2.1 Fotopletismografía

La fotopletismografía[7] es un método óptico simple, y de bajo coste, que permite detectar cambios de volumen sanguíneo en el lecho microcirculatorio de la piel. Se puede utilizar tanto para detectar la frecuencia cardiaca, como el flujo sanguíneo, o incluso la efectividad de la anestesia en una zona.

Habitualmente, para medir el pulso mediante fotopletismografía (ver Figura 2-2), se dirige un haz de luz a la piel y se miden los cambios en la cantidad de luz reflejada por la misma. Este es el principio en el que se basan los oxímetros, o los sensores de ciertos *smartphones*, para medir la frecuencia cardiaca o el nivel de oxígeno en sangre.

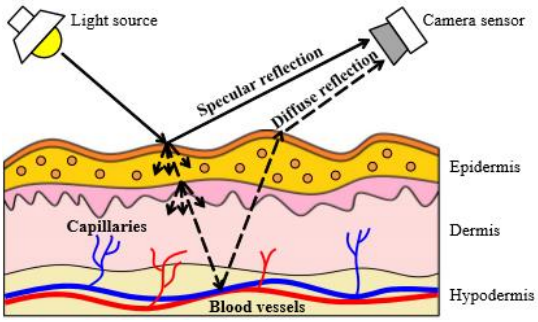

Figura 2-2 Funcionamiento de la fotopletismografía a distancia. [8]

Si bien, es posible realizar mediciones similares sin necesidad de emitir luz, simplemente utilizando una cámara y luz ambiente [9], ya que los cambios en los vasos sanguíneos de la piel producen cambios en la luz reflejada por la misma, los cuales quedan reflejados en los distintos canales de la imagen, con distinto peso en cada uno, puesto que la hemoglobina tiene una absorbancia variable entre el espacio visible y el infrarrojo. Estos métodos pueden tener problemas en su funcionamiento en función de las condiciones de luz.

Estos métodos consisten por lo general, en restringir la medición a una región de interés, analizar los componentes de la imagen por separado, procesarlos, y en el espacio de

frecuencias ver que frecuencia tiene la mayor cantidad de energía, siendo esta la frecuencia cardiaca detectada.

2.2.1.1 Algoritmos para estimar el pulso en un video mediante fotopletismografía

Uno de los primeros algoritmos para realizar fotopletismografía en remoto fue el publicado en 2008 por Verkruysse et al. [9]. Este algoritmo estima la frecuencia cardiaca mediante el siguiente proceso: en primer lugar, sobre un video de la cara de un sujeto, calcula las medias de color de cada canal RGB en una región de interés, generalmente la frente, en cada fotograma del vídeo. Estas medias son almacenadas en vectores, uno para cada canal RGB, sobre los cuales se realiza una Transformada Rápida de Fourier (FFT - *Fast Fourier Transform*), para posteriormente realizar un filtrado temporal haciendo uso de un filtro paso banda para descartar las frecuencias que no cuadren con las de una persona en reposo. Analizando las frecuencias de mayor energía en cada canal se obtiene una estimación del ritmo cardiaco.

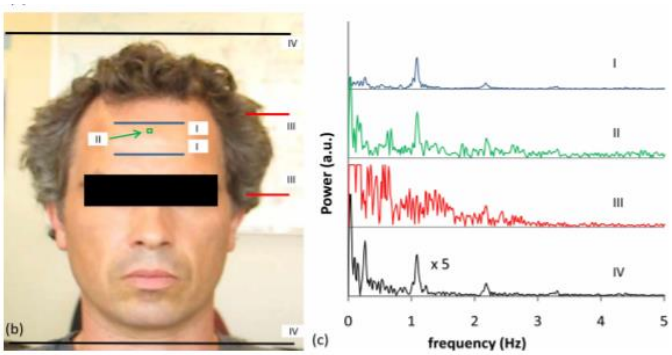

Figura 2-3 Energía de las frecuencias en cada canal [9]

En 2012, Wu et al. [10] desarrollaron un algoritmo basado en magnificación euleriana, que permitía amplificar pequeños cambios en objetos y personas, imperceptibles en principio para el ojo humano, y amplificarlos para que puedan ser observados a simple vista.

El funcionamiento de este algoritmo, de manera simplificada, es el siguiente: Se comienza generando una pirámide Laplaciana para cada fotograma del vídeo. Para cada nivel de la pirámide, se realiza un filtrado temporal para cada píxel a lo largo del tiempo mediante un filtro paso banda, que filtrará las frecuencias que serán amplificadas. En la siguiente etapa, se multiplica cada nivel filtrado de la pirámide por un factor de amplificación, que puede variar para cada nivel de la pirámide, y se suma con el nivel correspondiente original. Por último, se reconstruyen los fotogramas originales a partir de las pirámides ya amplificadas, y se obtiene el video magnificado.

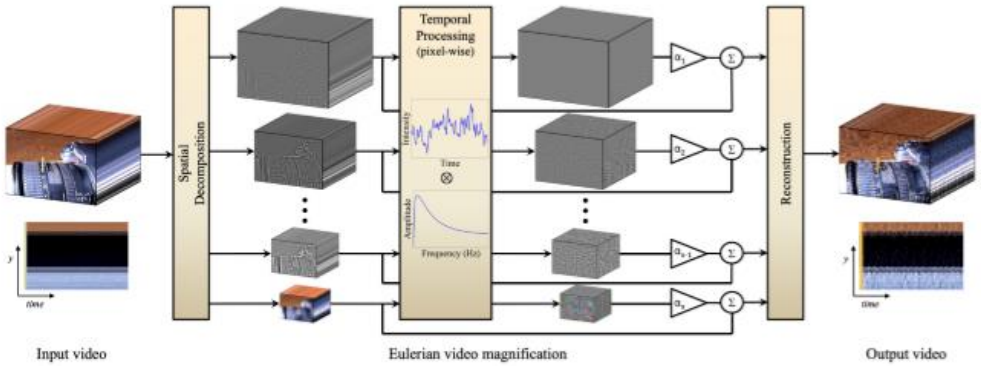

Figura 2-4 Proceso de magnificación Euleriana [10]

En 2018, Unakafov [11] publicó un estudio comparando el rendimiento y precisión de diferentes métodos para estimar la frecuencia cardiaca en remoto basados en fotopletismografía. La conclusión extraída es que la mejor estimación se obtenía siguiendo los siguientes pasos: se marca la cara entera como región de interés, y se utiliza un detector de piel para descartar los píxeles que no sean piel. Tras ello, se realiza un centrado en la media con un escalado sobre las muestras, un filtro de media móvil y un filtrado paso banda.
Tras ello, se obtiene la señal fotopletismográfica utilizando el método POS [8], pero se indica que ICA (*Independent Component Analysis*) y CHROM [12] también dan un buen resultado. Sobre esta señal se aplica otro filtro de media móvil y otro filtro paso banda, y por último un filtrado *wavelet,* y para estimar el pulso el método utilizado es una transformada *wavelet*.

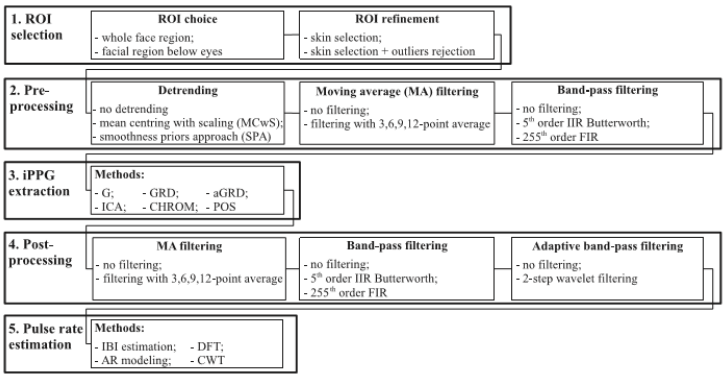

Figura 2-5 Esquema de los métodos considerados por Unakafov [11]

En el VPULab se cuenta además con diversos algoritmos para estimar la frecuencia cardiaca basados en fotopletismografía. Los algoritmos desarrollados por Ana Martín Doncel [13] hacen uso de la fotopletismografía en remoto y de la magnificación Euleriana [10], para obtener la frecuencia cardiaca a partir de un vídeo, el cual primero es magnificado para amplificar la variación del color de la piel, y después esta variación se mide para estimar el pulso. El algoritmo desarrollado por Julia Simón Chico [14], se basa en una modificación del algoritmo de fotopletismografía en remoto que trabaja sobre la media total de los colores en la imagen en lugar de separando por canales[15], y por último

el algoritmo desarrollado por Fernando Molina Sanz [16] sigue el algoritmo propuesto en Unakafov, pero utilizando CHROM[12] a la hora de obtener la señal fotopletismográfica en lugar de POS[8], y utilizando DFT para obtener la estimación de la frecuencia cardiaca.

2.2.2 Obtención del ritmo cardiaco mediante movimiento

En cada latido, el corazón bombea sangre a la cabeza a través de la arteria carótida. Este flujo de sangre produce una reacción Newtoniana que conlleva un pequeño movimiento de la cabeza, apenas perceptible para el ojo humano, pero medible para un ordenador.

2.2.2.1 Algoritmos para estimar el pulso a partir de un video basados en detección de movimiento

En 2013, Balakrishnan et al. [17], publicaron un artículo en el cual se detallaba un método para poder detectar la frecuencia cardiaca a partir de los movimientos involuntarios de la cabeza al recibir sangre desde el corazón.

Este algoritmo sigue el siguiente proceso: Se comienza localizando en el video la cara del sujeto, y se marcan como regiones de interés la frente y la zona bajo la nariz. Posteriormente se detectan puntos característicos, los cuales son seguidos utilizando el método de Lucas-Kanade [18], para obtener una serie de las posiciones en el tiempo para cada punto característico. Estas series con las posiciones son filtradas temporalmente mediante un filtro Butterworth restringiendo las frecuencias que queremos observar a valores factibles de frecuencia cardiaca. El siguiente paso consiste en realizar una descomposición PCA sobre las trayectorias filtradas, para después buscar la que tiene una frecuencia más clara y limpia en el espacio de frecuencias. Se localiza en esta señal la frecuencia de mayor energía, y esta frecuencia será la frecuencia cardiaca estimada.

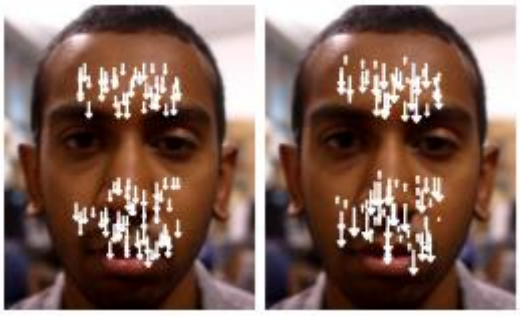

Figura 2-6 Magnitud de las trayectorias de los puntos característicos [17]

Este algoritmo funciona adecuadamente siempre que la cámara esté estática, si bien, al intentar realizar mediciones con este algoritmo desde un dispositivo móvil, como un *smartphone*, este algoritmo pierde precisión debido al movimiento de la mano. Una mejora de este algoritmo para poder ser ejecutado desde un dispositivo móvil se presenta en el trabajo realizado en 2019 por Lomaliza et al. [19], en el cual se hace uso de dos cámaras en un *smartphone* para mejorar la estimación del pulso, intentando detectar el movimiento del dispositivo al ser sostenido por el sujeto.

Para ello, el algoritmo realiza los siguientes pasos: La cámara frontal se utiliza para obtener imágenes de la cara y obtener una estimación siguiendo el algoritmo de Balakrishnan [17], mientras que la cámara trasera también recopila imágenes del entorno, en donde se buscan también puntos de interés, los cuales son también seguidos y se almacenan sus trayectorias, las cuales son analizadas, y utilizadas para filtrar las obtenidas por la cámara frontal, para reducir el ruido producido por el movimiento de la mano, y obtener una estimación de la frecuencia cardiaca más precisa.

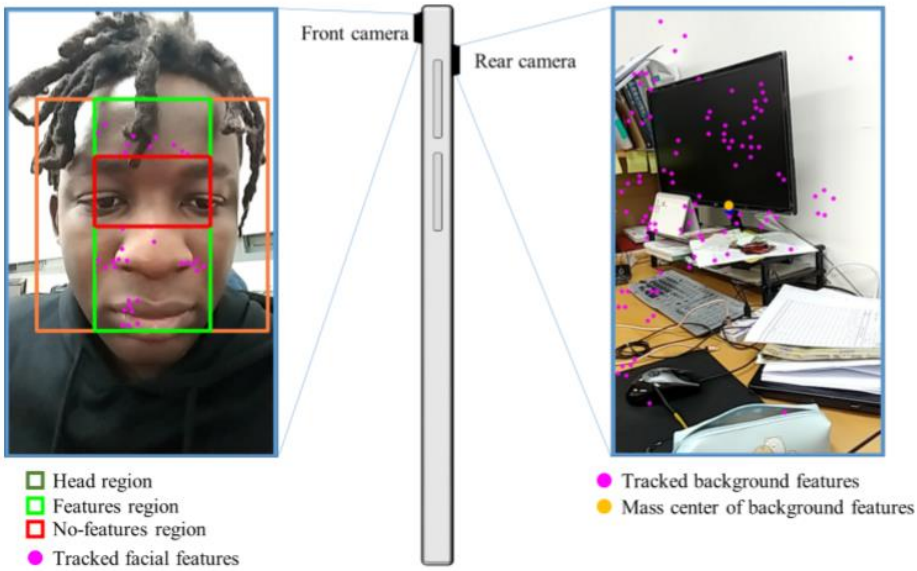

Figura 2-7 Funcionamiento del algoritmo de Lomaliza et al [19]

2.3 Pulsímetros Digitales Bluetooth para medir el ritmo cardiaco.

Hay muchos tipos de pulsímetros digitales, pero los más generalizados para usuarios no especializado son los pulsímetros de banda, que utilizan electrodos para medir el pulso, y los pulsímetros con sensores ópticos. Hoy en día, se pueden encontrar pulsímetros en una gran variedad de *wearables* como relojes electrónicos, ropa deportiva inteligente o dispositivos especializados pensados para deportistas.

Normalmente, estos pulsímetros cuentan con funcionalidad *Bluetooth* para poder sincronizar y compartir datos con los *smartphones* de sus usuarios, y poder llevar un registro de la frecuencia cardiaca, ya sea durante sesiones de ejercicio físico, a lo largo de todo el día o incluso mientras se duerme.
En este trabajo se utilizará el pulsímetro Polar H7, que se sitúa en el pecho y mide la frecuencia cardiaca utilizando una banda de electrodos.

3 Diseño

3.1 Introducción

Inicialmente se cuenta con unos algoritmos para estimar la frecuencia cardiaca a partir de secuencias de vídeo y una aplicación cuya finalidad es facilitar la implementación y testeo de nuevos algoritmos para estimar la frecuencia cardiaca.

Estos algoritmos se encuentran programados en Matlab, por lo que el primer objetivo de este trabajo es migrar estos algoritmos a un lenguaje en el cual puedan ser ejecutados haciendo uso de la librería OpenCV, para después añadir los algoritmos a la aplicación para poder realizar pruebas de manera sencilla desde la misma. En la sección 3.2 se detalla el funcionamiento de estos algoritmos, y las aproximaciones que se van a tomar para implementarlos en OpenCV.

Existen diferentes librerías y herramientas para visión por ordenador, pero en este trabajo se ha decidido utilizar OpenCV, por ser una librería de código con licencia BSD (*Berkeley Software Distribution*) la cual impone muy pocas restricciones a la hora de utilizar y distribuir el software licenciado.

La aplicación se encuentra desarrollada en C#, y en su estado actual cuenta con la funcionalidad de grabar videos almacenando también la frecuencia cardiaca del sujeto al que se graba, haciendo uso de un pulsímetro *Bluetooth*, en concreto el Polar H7. La aplicación también cuenta con un modo para reproducir un video previamente grabado, y aplicar un algoritmo de estimación de frecuencia cardiaca sobre el mismo, para contrastar la estimación del algoritmo con el pulso registrado desde el pulsímetro. En principio solo cuenta con un algoritmo integrado, basado en fotopletismografía.

Esta aplicación se pretende mejorar en diferentes aspectos. En el apartado de funcionalidad, el objetivo es implementar un módulo que permita realizar estimaciones en tiempo real permitiendo elegir el algoritmo con el que estimar. Además, se quiere optimizar el rendimiento actual de la aplicación a la hora de ejecutarse sobre videos pregrabados. Para conseguir estos objetivos se modificarán algunos de los módulos, principalmente el de algoritmos y el módulo principal ya existentes. Todos estos cambios, así como el estado actual de la aplicación, y el estado final que se espera conseguir se detallan en la sección 3.3

3.2 Migración Algoritmos

La primera parte de este trabajo ha consistido en estudiar la librería OpenCV y los algoritmos basados en color y luminancia [13] disponibles en Matlab para posteriormente migrarlos a C++, puesto que OpenCV está desarrollado de forma nativa en C++. El primer algoritmo que se migrará será el algoritmo de color, y posteriormente el de luminancia. El algoritmo basado en fotopletismografía [14] ya está disponible en C#, aunque también se han diseñado algunas mejoras para el mismo.

El algoritmo de Fernando [16] no ha sido implementado debido a que suponía más tiempo del correspondiente a este Trabajo de Fin de Grado.

3.2.1 Funcionamiento Algoritmo de Color

El diagrama de flujo general del algoritmo de color está representado en la siguiente imagen:

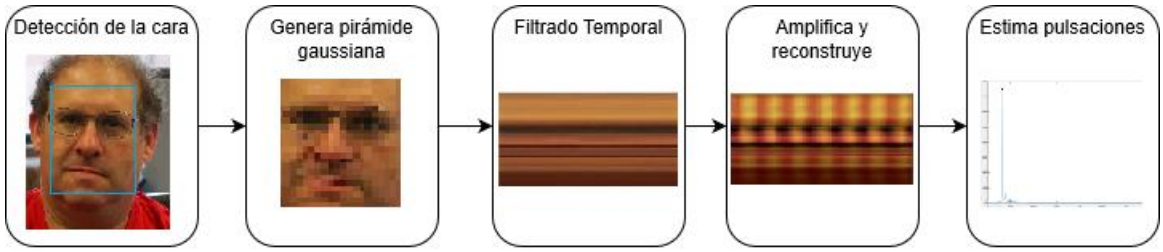

Figura 3-1 Esquema del funcionamiento del algoritmo de color

A continuación, se detalla el diseño y funcionamiento de cada uno de estos bloques.

3.2.1.1 Detección de la Cara

En el algoritmo en Matlab se detecta la cara en el primer fotograma, y la región que haya detectado se utiliza en el resto de los fotogramas, pues se asume que el sujeto no se mueve más de lo normal cuando se está mirando una pantalla, o durmiendo.

Esta detección se realizará mediante un detector en cascada, el cual realiza varios barridos a la imagen para clasificar adecuadamente el objeto buscado, y un clasificador que sigue el algoritmo de Viola-Jones [20].

Hay otros clasificadores que se pueden utilizar para detectar caras, como *Local Binary Patterns* (LBP) [21], en la parte de desarrollo se realizan pruebas con estos clasificadores y se valora si conviene utilizarlos en lugar de Viola-Jones.

Para intentar aumentar la precisión del algoritmo, se ha valorado la posibilidad de realizar un seguimiento de la cara, para intentar eliminar el ruido inducido por el movimiento del sujeto. Este seguimiento podría consistir en detectar la cara en todos los fotogramas en lugar de únicamente el primero, o en realizar una detección, y luego un seguimiento utilizando el algoritmo KLT [18].

Si bien, al buscar información sobre esta segunda posibilidad, se ha llegado a la conclusión de que esto genera dos posibles fuentes de ruido. La primera de ellas es que la luz que incide sobre la región de interés puede variar al moverse el sujeto, afectando a la medición. Además, el propio proceso de seguimiento KLT puede introducir ruido si la región se está moviendo de manera constante.

Por este motivo se ha decidido mantener el modelo de detectar una cara, y recortar la región encontrada en todos los fotogramas. Ver la sección 5.2.1.2 para una comparativa entre métodos.

3.2.1.2 Generar la pirámide Gaussiana

El siguiente paso es generar una pirámide gaussiana de *n* niveles, aplicando un filtro binomial en cada etapa, para obtener el nivel más alto de la misma, que nos dará una imagen de menor resolución que podremos procesar más rápido en la siguiente etapa.

3.2.1.3 Filtrado Temporal

El siguiente paso consiste en agrupar todas las imágenes obtenidas al generar las pirámides gaussianas de cada fotograma y agruparlas en una matriz de cuatro dimensiones, siendo las dos primeras dimensiones el alto y el ancho de los fotogramas, la tercera dimensión el canal de la imagen (rojo, azul o verde), y la última dimensión el tiempo, o número de fotograma.

El filtrado temporal se puede esquematizar en el siguiente diagrama, que representa los pasos a seguir para filtrar temporalmente cada canal.

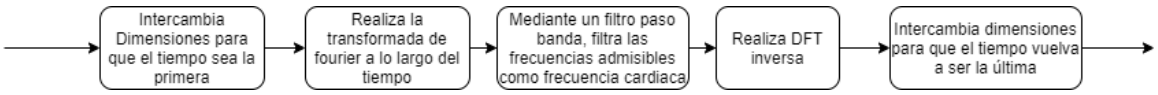

Figura 3-2 Pasos del filtrado temporal

Antes de filtrar temporalmente, es necesario intercambiar las dimensiones (*shiftdim* en Matlab) para que el tiempo sea la primera, puesto que queremos filtrar temporalmente, será necesario tener vectores/imágenes de las cuales el tiempo sea una de las dimensiones, en concreto se utilizarán imágenes con dimensiones Alto x Tiempo. Esta operación presenta la primera limitación que encontramos con OpenCV, ya que mientras que Matlab puede manejar un video como una matriz 3-dimensional, OpenCV no soporta este tipo de operaciones, y un video se representa como una lista de fotogramas.

Para realizar el filtrado se utiliza un filtro paso banda ideal, ya que solo nos interesa amplificar las variaciones de color con frecuencias admisibles como pulso cardiaco, y descartar el resto.

Una vez filtrado el video, ha de volver a intercambiar dimensiones para tener de nuevo una matriz de Alto x Ancho x Canal x Tiempo.

3.2.1.4 Amplifica y reconstruye

En el algoritmo original, se redimensiona la señal filtrada, se multiplica por un factor de amplificación y se suma a la señal original para obtener la señal reconstruida y amplificada. En el siguiente paso se obtienen las medias de color de esta imagen.

Para el funcionamiento del algoritmo no es necesaria la señal reconstruida, siendo necesaria únicamente si se va a mostrar el resultado de la amplificación. Como redimensionar una imagen puede resultar costoso computacionalmente, se ha optado por calcular las medias de cada color de la imagen filtrada, multiplicarlas por el factor adecuado, y sumarlas a las medias de color originales, en lugar de calcular la media de la suma de la imagen amplificada y la original.

3.2.1.5 Obtener frecuencia cardiaca

Una vez que se ha amplificado y reconstruido el video, se procede a extraer la frecuencia cardiaca. En el algoritmo de color se realizan las mediciones del pulso centrándonos en 3 regiones de interés diferentes: la cara al completo, la frente, y el espacio que hay bajo la nariz.

Para estimar el pulso se sigue el siguiente algoritmo en cada ROI:

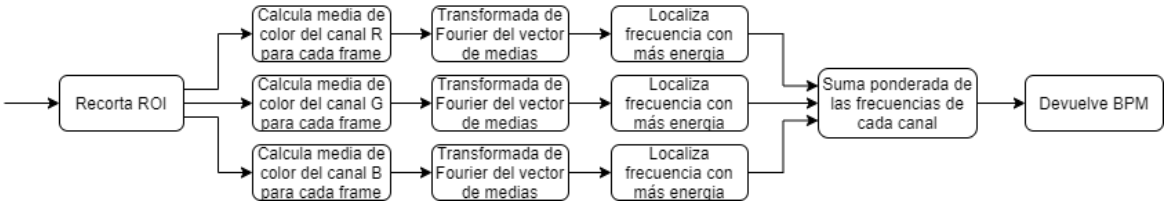

Figura 3-3 Diagrama de flujo para el cálculo de la frecuencia cardiaca en el algoritmo de color.

Para que resulte más preciso, a la hora de localizar la frecuencia con más energía, solo se tendrán en cuenta las frecuencias posibles y normales, en un rango de 40 a 180 pulsaciones.

3.2.2 Funcionamiento Algoritmo de Luminancia

La luminancia es una medida fotométrica de la intensidad de la luz por unidad de área de luz viajando en una dirección determinada. Describe la cantidad de luz emitida o reflejada en un área en concreto.

El espacio de color *YCbCr* se compone de tres canales, siendo *Y* el canal de la luminancia, *Cb* el canal de la diferencia de azules y *Cr* el canal de la diferencia de rojos. Como la luminancia nos da información sobre la cantidad de luz emitida o reflejada en un área, y por lo comentado en el capítulo 2.2.1, se puede utilizar para medir la frecuencia cardiaca.

El algoritmo sigue el siguiente esquema:

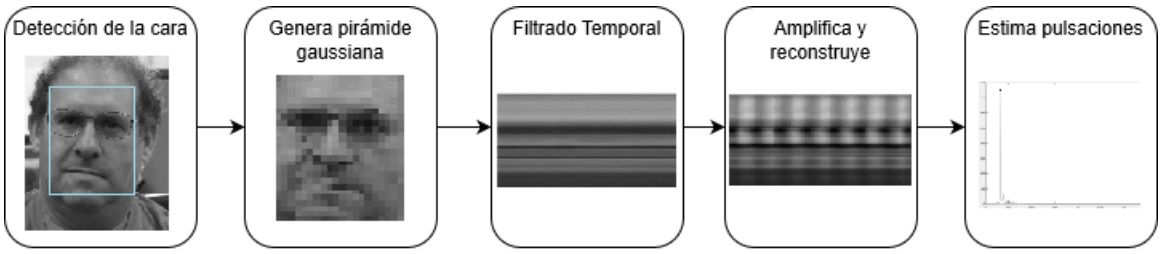

Figura 3-4 Esquema del funcionamiento del algoritmo de la luminancia

Este algoritmo es muy similar al anterior en funcionamiento, pero trabaja con un único canal en lugar de 3. Por este motivo, el diseño de las etapas apenas varía respecto al algoritmo de color, y en la etapa de detección de la cara se mantiene de hecho idéntico. A continuación, se detallan los cambios de diseño en el resto de etapas.

3.2.2.1 Generar la pirámide gaussiana

En esta etapa primero se convierte la imagen del espacio de color RGB al espacio de color *YCrCb*, para posteriormente realizar la pirámide gaussiana sobre el canal Y de la luminancia. De nuevo, se trabajará sobre el nivel más alto de la pirámide (la imagen de menor resolución).

3.2.2.2 Filtrado Temporal

El intercambio de dimensiones se realiza de manera idéntica al algoritmo de color, la diferencia es que la información que contiene cada píxel es un único valor (frente a tres de la imagen RGB).

El proceso del filtrado temporal es de nuevo idéntico al anterior, con la única diferencia de que ahora se trabaja con imágenes con un canal, y por tanto no es necesario separar canales antes de realizar el filtrado temporal, ni combinarlos de nuevo en una imagen tras el mismo.

3.2.2.3 Amplificación y Reconstrucción

A la hora de amplificar para la luminancia, se tiene de nuevo un factor de amplificación, pero ahora además hay otro factor, en este caso de atenuación, que se aplica sobre los canales de crominancia de la imagen.

De nuevo, para reconstruir la imagen primero se redimensionan los fotogramas ya filtrados y amplificados, y se suman al canal Y de la imagen original, puesto que es el canal sobre el que estamos trabajando.

3.2.2.4 Estimar frecuencia cardiaca

Para realizar la estimación, el proceso es similar al algoritmo de color, pero con la diferencia de que ahora se realiza el proceso sobre el canal de la luminancia, descartando los canales de contraste azul y contraste rojo.

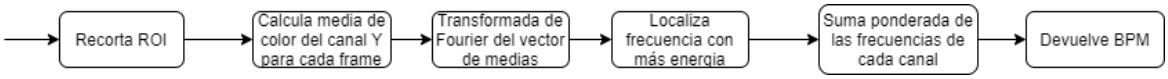

Figura 3-5 Proceso para estimar la frecuencia cardiaca desde el canal de luminancia

3.2.3 Comparativa Algoritmos Color y Luminancia

La principal ventaja que ofrece el algoritmo de Luminancia frente al algoritmo de color es que el procesamiento se realiza sobre un único canal, el canal Y de luminancia del espacio de color YCrCb, lo cual ofrece una mayor rapidez computacional respecto al algoritmo de color.

Si bien, el canal de luminancia solo contiene parte de la información de color, por lo que la que se está perdiendo algo de la misma, la cual no se pierde al utilizar el algoritmo de color, que funciona sobre RGB. Esta pérdida de información puede suponer que en algunos casos la estimación del algoritmo de luminancia sea peor que el de color.

Para más información sobre las diferencias entre estos algoritmos se puede consultar el trabajo realizado por Ana Martín Doncel [13].

3.2.4 Mejoras al algoritmo PPG de la aplicación

La aplicación ya existente, descrita en más detalle en la sección 3.3, dispone ya de un algoritmo basado en pletismografía, si bien el tiempo de ejecución del mismo resulta demasiado elevado (9.5 segundos para un vídeo de 3,8 segundos -115 fotogramas, a 30 fotogramas por segundo- con resolución 1920 x 1080 píxeles).

Este rendimiento no lo haría apto a priori para su ejecución en tiempo real, por lo que se ha procedido a analizar que posibles mejoras se le pueden hacer para mejorar estos tiempos.

Analizando el funcionamiento de este algoritmo, se deduce que sigue el siguiente diagrama de flujo:

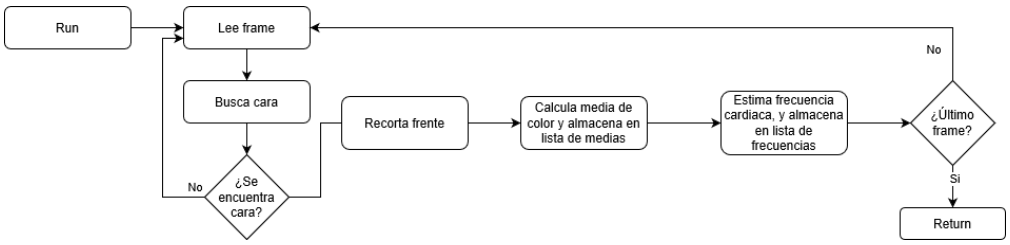

Figura 3-6 Diagrama de flujo del PPG actual

Se ha diseñado una nueva versión de este algoritmo con la finalidad de optimizarlo para poder ser utilizado en tiempo real. El nuevo diagrama de flujo quedaría así:

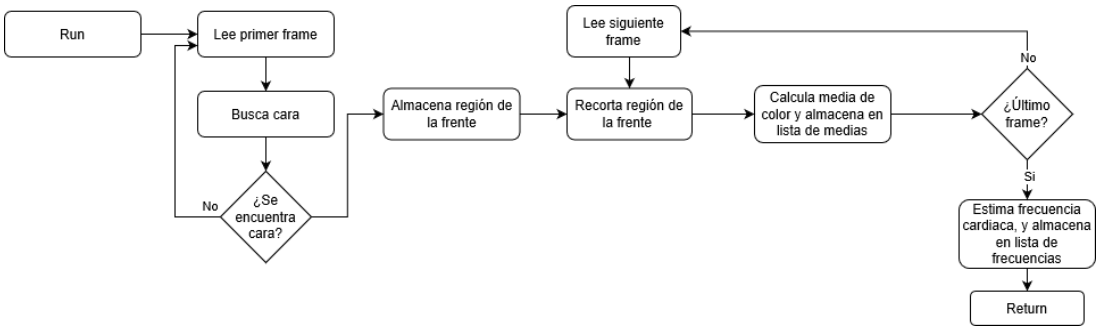

Figura 3-7 Diagrama de flujo del PPG mejorado

De esta forma se reduce el número detecciones de la cara realizados, ya que estos son computacionalmente pesados, y puesto que la persona a la que se le va a realizar la medición debe estar estática, se puede asumir que la frente va a estar siempre en la misma región del fotograma.

En este diseño se disminuye también la cantidad de estimaciones realizadas al ejecutar el algoritmo, puesto que la medición que interesa es para toda la ventana de tiempo (todos los fotogramas).

El desarrollo de estas mejoras se ha detallado en la sección 4.3 de Integración de Algoritmos.

3.3 Estado actual de la aplicación

En primer lugar, se analiza cual es el estado inicial de la aplicación, para evaluar adecuadamente los cambios a realizar, y como incorporar el módulo nuevo de manera adecuada.

La aplicación en su estado inicial consta de cuatro módulos (ver Figura 2-1): El módulo *principal* se encarga de manejar la interfaz, el módulo *pulsímetro* se encarga de conectarse con un pulsímetro *Bluetooth* (modelo Polar H7) y obtener una medición, el módulo

algoritmos es el encargado de manejar los algoritmos disponibles, y el módulo de *grabación* que se encarga de gestionar la grabación de videos mediante Kinect.

El módulo principal contiene la lógica de la interfaz, y se encarga de manejar el resto de los módulos para que puedan ser utilizados.

La funcionalidad proporcionada por el módulo de grabación es la siguiente: Se puede grabar un video utilizando una Kinect mientras se mide el pulso utilizando un pulsímetro *Bluetooth*, y registrar también el pulso medio durante la grabación del vídeo para su posterior análisis. Estos videos pueden grabarse como imágenes de color y como imágenes de profundidad.

Por otro lado, en el módulo de algoritmos se dispone de un algoritmo basado en fotopletismografía ya implementado, el cual se puede utilizar desde la pestaña de probar algoritmos, implementada en el módulo principal, para estimar el pulso a partir de uno de los videos grabados previamente con el módulo de grabación.

También se dispone de un módulo pulsímetro, el cual es capaz de obtener la frecuencia cardiaca a partir de un dispositivo Polar H7 mediante *Bluetooth*, y mostrarla por pantalla actualizándose al recibir una nueva medición.

Estos módulos y sus relaciones se pueden visualizar en el siguiente diagrama:

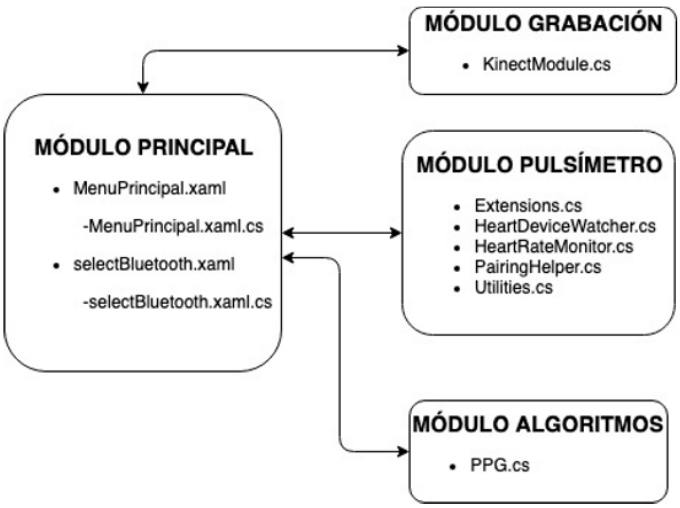

Figura 3-8 Diagrama general de la aplicación [14]

La interfaz en su estado actual ofrece dos modos diferentes. El primero de ellos es el modo *Grabar*, que hace uso del módulo de grabación, en el cual se pueden ajustar parámetros como la duración del video y el nombre que van a tener los ficheros generados. También permite seleccionar el dispositivo *Bluetooth* a utilizar para obtener la frecuencia cardiaca, aunque por el momento solo está disponible el dispositivo Polar H7.

El segundo modo, el modo *Algoritmos*, permite seleccionar un video y un algoritmo para intentar estimar la frecuencia cardiaca del sujeto en el video utilizando el algoritmo seleccionado. Este modo será renombrado a *Fichero* en la versión final de la aplicación.

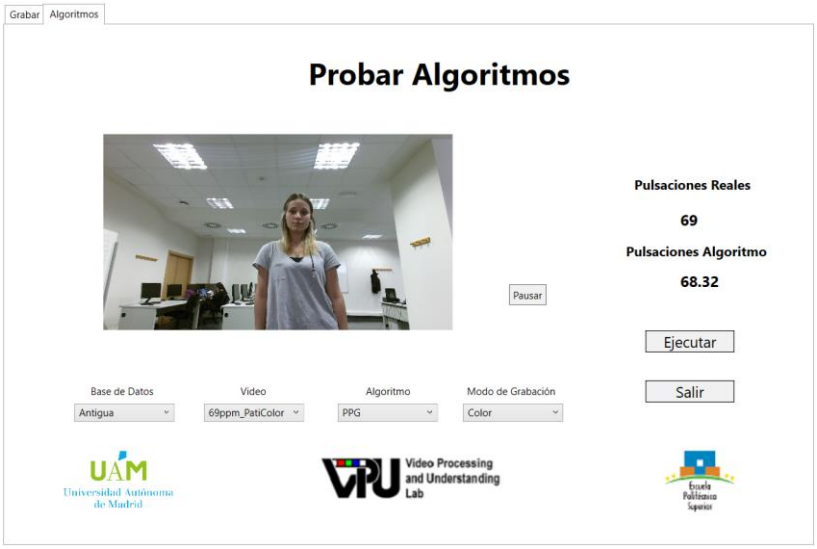

Figura 3-9 Interfaz actual

Se ha realizado una ejecución de prueba sobre uno de los videos almacenados en la aplicación para comprobar el rendimiento, y se ha registrado un tiempo de ejecución del algoritmo PPG, para un video de 3,8 segundos (115 imágenes a 30 fotogramas por segundo) con resolución 1920 x 1080 píxeles, es de 9.5 segundos en promedio (para 10 ejecuciones).

3.4 Modificaciones y mejoras

En primer lugar, se creará un módulo para el análisis de video en tiempo real, el cual permitirá mediante una *webcam* realizar estimaciones de frecuencia cardiaca a partir de un video en tiempo real, y además permitirá contrastar estas estimaciones con el pulso real obtenido a través del pulsímetro *Bluetooth*.

También se rediseñará el módulo de algoritmos, para seguir un sistema de herencia de clases, y además preparar la ejecución de estos algoritmos para ser paralelizables, y de esta forma evitar perder rendimiento en el hilo de la interfaz.

Con estos cambios, el diagrama general de la aplicación pasaría a tener la siguiente forma:

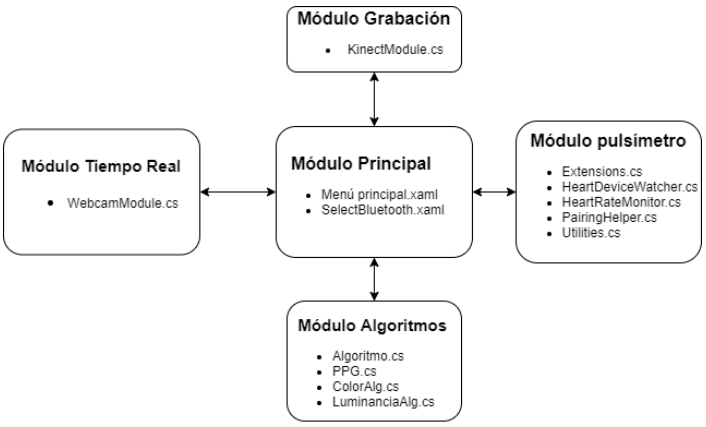

Figura 3-10 Diagrama general de la aplicación final

El diseño del módulo en tiempo real se detalla en la sección 3.4.1, y los cambios realizados en el módulo de algoritmos se detallan en la sección 3.4.2.

Otra de las mejoras que se implementarán será una forma de visualizar los diferentes valores que procesan los algoritmos, para poder estudiar su funcionamiento y depurarlos de manera más sencilla. Esto se puede realizar añadiendo por ejemplo una gráfica a la pestaña de prueba de algoritmos.

3.4.1 Diseño del módulo de la aplicación para analizar video en directo.

A la hora de diseñar el módulo para extraer la frecuencia cardiaca en tiempo real se han tenido en cuenta algunos requisitos previos:

- El módulo debe responder en tiempo real, tanto a la entrada del usuario como a la hora de mostrar las imágenes que se toman desde la *webcam*.
- Debe ser capaz de mantener los tiempos de respuesta independientemente de lo computacionalmente pesado que resulte el algoritmo que se esté ejecutando en cada instante. Únicamente debería verse afectada la frecuencia con la que se muestran nuevas estimaciones del pulso.
- Debe ser mantenible.

Para poder asegurar que la aplicación es responsiva, se ha decidido desarrollar el módulo de la *webcam* para ejecutarse en un hilo aparte. El diagrama de flujo del hilo principal de la aplicación será por tanto el siguiente:

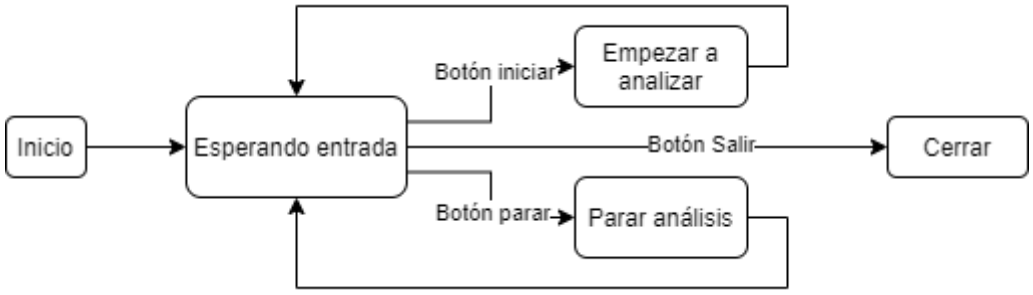

Figura 3-11 Diagrama de flujo del modo Tiempo Real

Al pulsar el botón de iniciar, se lanzará el segundo hilo, cuyas funciones principales serán preparar las imágenes para ser procesadas por el algoritmo seleccionado, mostrar las imágenes que toma la *webcam* en una ventana y actualizar una gráfica que contrasta la frecuencia registrada mediante un pulsímetro y la frecuencia estimada por el algoritmo que se esté ejecutando.

Este hilo se encargará además de ir almacenando los fotogramas en la cola de ejecución del algoritmo, para que sean procesados cuando el algoritmo se ejecute. El diagrama de flujo de este hilo es el representado en la figura 3-9.

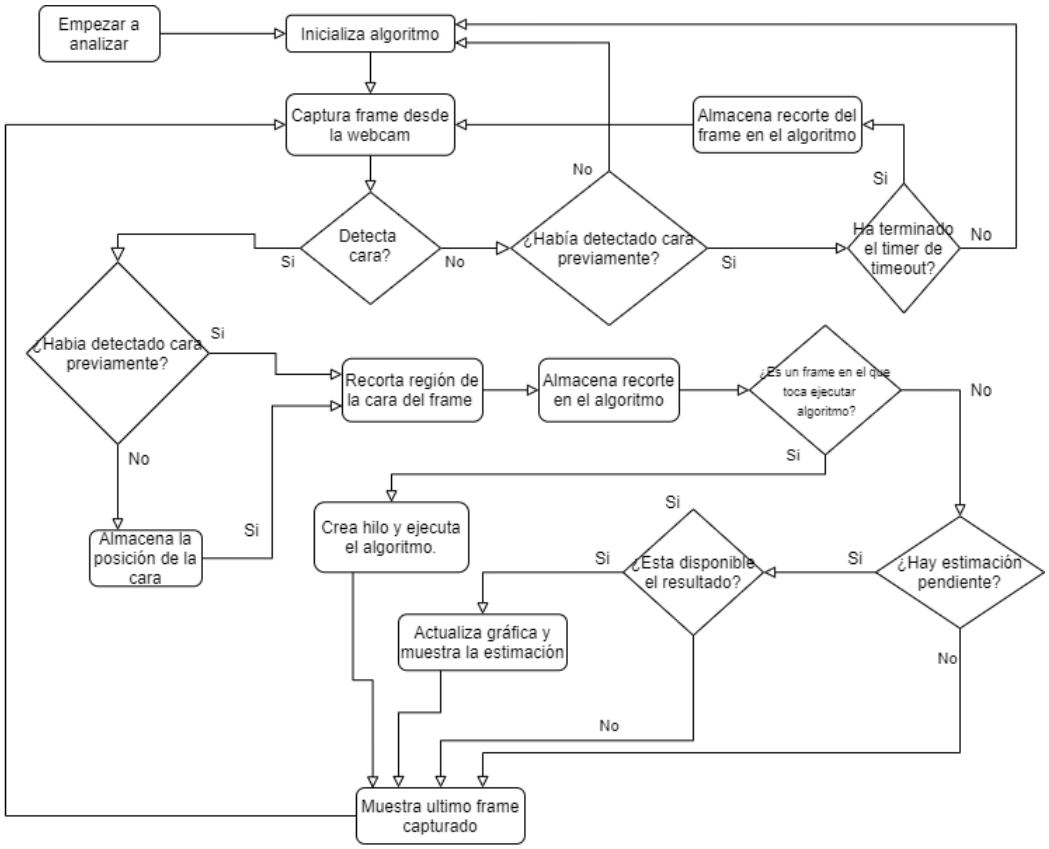

Figura 3-12 Diagrama de flujo del hilo que gestiona la *webcam*

El tercer hilo se encargará de ejecutar el algoritmo sobre las imágenes que se han almacenado.

Si en la aplicación se pulsa el botón parar, se dejarán de realizar nuevas capturas de imágenes y nuevas mediciones, pero se mantendrá la gráfica con los datos que contenga para poder contrastarlos. Se terminarían los otros dos hilos de estar funcionando.

3.5 Integración de los algoritmos en la aplicación.

Para poder facilitar la integración de nuevos algoritmos en la aplicación, se ha decidido tomar una aproximación de herencia de clases en el módulo de algoritmos, y prepararla para que se pueda ejecutar en paralelo con los otros hilos de la aplicación, para de esta forma poder soportar algoritmos más pesados computacionalmente sin afectar al rendimiento de la interfaz de la aplicación.

3.5.1 Diseño de la clase algoritmo

Para integrar los algoritmos dentro de la aplicación, se han tenido en consideración los requisitos básicos que deberían satisfacer:

- Los algoritmos deben ser capaces de funcionar recibiendo únicamente la entrada de las imágenes a procesar, y la orden de ejecutarse.
- Deben ser mantenibles.
- Deben ser compatibles tanto con ejecuciones a partir de un video en memoria, como a partir de un video en tiempo real.

Por este motivo, la aproximación tomada ha sido un sistema de herencia, teniendo una clase abstracta que represente un algoritmo, y los algoritmos que se añadan la extiendan. En el diagrama siguiente se representan esta clase abstracta, con sus propiedades más relevantes y los métodos que todo algoritmo que herede de la misma tendrá que implementar.

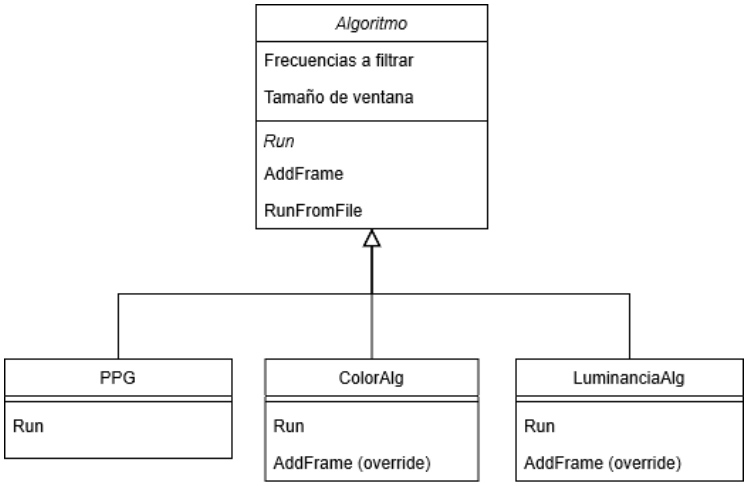

Figura 3-13 Diagrama de clases para el módulo de algoritmos

El método virtual *AddFrame* se encargará de añadir un fotograma al algoritmo. Por defecto, esta función añade los fotogramas a una lista de imágenes pendientes de procesar que tendrán todas las implementaciones de los algoritmos, aunque por ser un método virtual podrá ser sobrescrito en cada uno de los algoritmos para almacenar las imágenes en las colas que más convenga, o incluso para realizar algún tipo de procesamiento sobre las mismas antes de ser analizadas por el algoritmo. Esta función se ejecuta para cada fotograma que se añade al algoritmo, por lo que será necesario que el tiempo de procesamiento sea lo suficientemente reducido como para mantener la visualización de imágenes en tiempo real a la vez que se añaden los fotogramas.

El método abstracto *Run*() se utiliza para ejecutar el algoritmo sobre los fotogramas que se le hayan añadido previamente mediante el método *AddFrame*. Al terminar esta ejecución, y si no ha habido ningún error, se cambiará a verdadero el valor de un *flag* que indicará que hay una estimación disponible para ser leída, y una propiedad que almacenará el resultado de dicha estimación.

Por último, el método *RunFromFile()* se encargará de cargar los fotogramas de un video desde el disco (siempre que el algoritmo se haya inicializado con una ruta de video), y de ejecutar el algoritmo sobre dichos fotogramas.

Además, la clase algoritmo tendrá también como propiedades unas listas pensadas para almacenar cada una de las estimaciones realizadas por el algoritmo, así como valores que puedan resultar interesantes de observar, ya sea para su estudio posterior o para facilitar la depuración del algoritmo, como las medias de los colores de cada una de las imágenes, o el resultado de cada una de las transformadas de Fourier sobre los vectores de las medias de imágenes.

4 Desarrollo

La primera parte del desarrollo ha consistido en utilizar la librería OpenCV para reproducir el resultado del código de Matlab desde C++, puesto que es el lenguaje en el que está programado OpenCV de forma nativa. OpenCV dispone de interfaces para ser utilizada desde Python, Java y Matlab, y existen *wrappers* para manejarla desde otros lenguajes, como *EmguCV* que permite utilizar OpenCV desde C#. También se han realizado cambios sobre el algoritmo ya integrado en la aplicación para mejorar su precisión y rendimiento.

La segunda parte del desarrollo ha consistido en implementar el módulo de Tiempo Real en la aplicación actual, y en realizar las modificaciones adecuadas a la aplicación para facilitar su paralelización y la implementación de nuevos algoritmos.

Por último, se han integrado en la aplicación los algoritmos ya migrados a OpenCV, para que puedan ser utilizados tanto en tiempo real como sobre videos previamente grabados.

La versión de OpenCV en el momento de realizar este trabajo, y con la que se va a trabajar a lo largo del mismo desde C++ es la versión 4.2.0. Para el trabajo en C# se utilizará *EmguCV* en su versión 3.4.3.

4.1 Migración de los Algoritmos

El proceso para migrar los algoritmos desde Matlab a OpenCV seguido ha consistido en replicar los pasos realizados en el código de Matlab, pero utilizando las funciones disponibles en OpenCV y del lenguaje C++. Algunas funciones de Matlab no tienen un equivalente en OpenCV, por lo que ha sido necesario implementarlas, o en alguna ocasión, buscar alguna alternativa.

4.1.1 Algoritmo de Color

A la hora de migrar el algoritmo de color se ha intentado seguir lo más posible el código de Matlab, aunque se ha encontrado algún problema debido a funciones no disponibles en OpenCV, y con la forma de manejar los datos de OpenCV. Las soluciones utilizadas para solventar estos problemas, y las funciones utilizadas para replicar el código, se detallan en este capítulo.

4.1.1.1 Detección de la cara

Para realizar la primera etapa, detección de la cara, en Matlab se utiliza un *CascadeObjectDetector*, que tiene su equivalente en OpenCV en la clase *CascadeClassifier* de OpenCV. Llamando al método *DetectMultiScale* realizamos la detección en cascada sobre un fotograma. Esta función recibe como parámetro la imagen sobre la que detectar características, y un clasificador en cascada.

El funcionamiento de estos clasificadores consiste en aplicar clasificadores cada vez más precisos a las regiones de interés de la imagen de manera consecutiva, con lo que los clasificadores más costosos computacionalmente solo se aplican a las regiones que han pasado una clasificación previa.

El clasificador utilizado para detectar la cara ha sido un clasificador en cascada Haar basado en el algoritmo de Viola Jones [20]. Uno de los problemas que tienen estos clasificadores es que el rendimiento puede empeorar considerablemente si se detectan muchos falsos positivos, ya que al ser analizados varias veces los mismos hasta que se descartan, consumen un tiempo de computación considerable.

Por este motivo, se han buscado alternativas que disminuyesen los posibles problemas de rendimiento, para que a la hora de utilizar los algoritmos en tiempo real fuesen lo más responsivos posibles.

En primer lugar, se probó a utilizar un clasificador LBP en lugar de Haar. Los patrones binarios locales (LBP) son operadores de texturas que catalogan los pixeles de una imagen aplicando un umbral a los píxeles vecinos con el valor del píxel del centro, y considera el resultado un número binario. Tiene la ventaja de ser más ligero computacionalmente que un clasificador Haar, pero a expensas de ser menos preciso. Tras realizar diferentes pruebas sobre fotogramas de diferentes videos, se llegó a la conclusión de que, pese a ser por lo general más rápido, en algunos fotogramas podía ser también más lento, por lo que se descartó esta posibilidad.

La otra medida estudiada, y la que se decidió adoptar, es limitar los tamaños máximos y mínimos de los objetos que puede detectar el clasificador. Esto nos permite realizar barridos mucho más eficientes incluso en casos en los que el detector encontraba bastantes falsos positivos que descartar.

Para encontrar los tamaños máximos y mínimos adecuados, en primer lugar se detectan las caras en una imagen con un tamaño mínimo pequeño, de 30 por 30 píxeles, ya que un rostro menor que eso en tamaño no nos va a aportar información suficiente para ejecutar el algoritmo, y tras detectar la cara, se ajustan el tamaño máximo y mínimo al tamaño de la cara más menos un margen arbitrario, puesto que el sujeto puede moverse ligeramente y el propio clasificador puede, en dos fotogramas consecutivos, y sin apenas cambios, detectar la cara en regiones de tamaños ligeramente diferentes.

Para calcular la ROI correspondiente a la frente tras detectar la cara, se realizan recortes en la imagen que se ha obtenido de la cara. Para ello, nos quedamos con el 25% central del ancho de la cara, y con el segundo quinto comenzando por la parte superior de la cara. Para recortarla, basta con crear un nuevo *Mat* utilizando como parámetros en el constructor la imagen original y un rectángulo indicando la región de interés que recortar.

4.1.1.2 Generar pirámide Gaussiana

Para generar la pirámide gaussiana desde Matlab se ha hecho uso de la librería externa *pyrTools*, pero esto es algo que OpenCV soporta de forma nativa haciendo uso de la función de *BuildPyramid*, la cual recibe una imagen, la cantidad de niveles y como gestionar el borde al reescalar, y devuelve un vector de imágenes con los diferentes niveles de la pirámide. En concreto, se trabaja sobre el más alto de la misma, es decir, la imagen de menor resolución. El filtro utilizado al generar los niveles es un filtro gaussiano.

En esta etapa se han encontrado dos problemas, el primero relacionado con el filtrado utilizado al generar la pirámide, y otro de rendimiento al generar la pirámide.

En el algoritmo original, el filtro utilizado al generar la pirámide es un filtro bilineal. OpenCV no soporta este filtro de forma nativa, y tampoco soporta generar pirámides con un filtro concreto. Una alternativa era crear el filtro binomial, y generar la pirámide filtrando y reescalando cada fotograma independientemente, pero resultaba más costoso computacionalmente. Por este motivo se ha decidido utilizar la función *BuildPyramid*, pese a que utilice un filtro gaussiano en lugar de binomial.

El problema de rendimiento se presenta al generar las pirámides gaussianas para cada capa de color por separado. Las pruebas de rendimiento se han realizado utilizando la librería *chrono* de C++. Se han realizado pruebas para dos videos de ejemplo del MIT, siendo el primero de ellos, *face.mp4*, un video de 300 fotogramas con una resolución de 528 x 592, y el segundo de ellos, *face2.mp4*, también de 300 fotogramas, pero con una resolución de 570x718 píxeles. Ambos métodos se han ejecutado un total de 10000 veces, los tiempos medios de ejecución se muestran en la siguiente tabla:

Video	*Tiempo ejecución canales de color juntos (µs)*	*Tiempo ejecución canales de color por separado (µs)*
face.mp4	55.56	209.58
face2.mp4	91.100	220.85

Tabla 4-1 Comparativa de rendimiento entre ambos métodos para generar las pirámides gausianas

Tras realizar la comparativa se ha decido realizar la pirámide con los canales juntos, y separarlos posteriormente para realizar el filtrado temporal, puesto que resulta más eficiente.

4.1.1.3 Filtrado Temporal

El filtrado temporal se puede esquematizar en el siguiente diagrama, que representa los pasos a seguir para filtrar temporalmente cada canal.

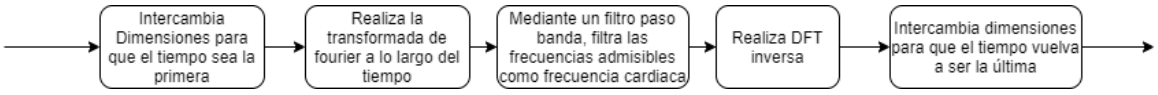

Figura 4-1 Diagrama de flujo para el filtrado temporal

Para intercambiar dimensiones en Matlab se utiliza la función *shiftdim*. Esta es la primera limitación de OpenCV con la que nos encontramos, pues pese que Matlab puede manejar un video como una matriz 4D de imágenes, con dimensiones Alto x Ancho x Canal x Tiempo, OpenCV no soporta los videos de esta manera, por lo que se ha tenido que trabajar la manera de cambiar las dimensiones para tener fotogramas temporales o *slices* que poder filtrar temporalmente.

Para ello, se ha creado una función, ShiftDim3C, que recibe un vector de imágenes RGB, y forma *slices* temporales, creando píxel a píxel la imagen temporal, cogiendo la información para cada pixel a lo largo del tiempo, y colocándola en su lugar correspondiente en una imagen Alto x Tiempo x Ancho.

En la siguiente imagen se muestra una representación del video como un volumen Alto x Ancho x Tiempo, y los *slices* que se obtendrían al cortar por las líneas verdes y rojas dicho volumen.

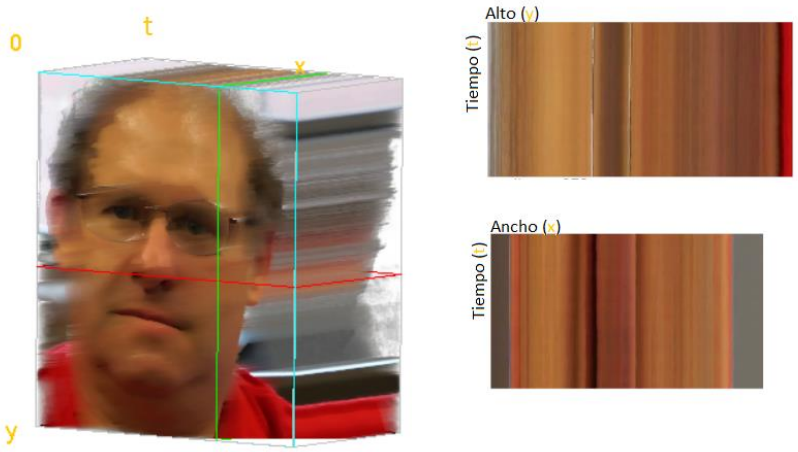

Figura 4-2 Visualización de *slices* espaciotemporales.

Realizar este cambio de dimensiones resulta computacionalmente costoso, pues se tiene que iterar sobre cada píxel de cada fotograma del video.

Al mantener el diseño del algoritmo original, tendríamos que realizar este cambio de dimensiones una vez por canal de color, pero resulta mucho más eficiente en términos de rendimiento realizar el cambio de dimensiones directamente sobre la imagen RGB, y separar posteriormente los canales para el filtrado temporal de cada uno de ellos.

Esto se debe a que al hacer el cambio de dimensiones para cada canal, se ejecutan tres bucles, con tantas iteraciones como píxeles haya, para copiar un único valor *float* por píxel (un bucle por canal), mientras que si se realiza sobre una imagen RGB canales, la información que se copia es directamente un vector con tres valores que representan la información de color de cada canal para ese píxel, y es suficiente con realizar un único bucle.

El siguiente paso es separar los canales de color para poderlos filtrar por separado, ya que la DFT se realiza en imágenes de un canal. Esto se consigue mediante la función *Split* de OpenCV. Este paso no es necesario en Matlab pues es capaz de realizar una FFT directamente sobre una matriz 4D.

En Matlab, al hacer una FFT (*Fast Fourier Transform*) se procesan los vectores de la matriz por separado, pero en OpenCV la función DFT (*Direct Fourier Transform*) realiza una transformada de Fourier directamente en 2 dimensiones. Para poder replicar el funcionamiento del código de Matlab y contrastar más fácilmente los resultados, se ha forzado a realizar la DFT en OpenCV con el *flag DFT_ROWS*, que permite calcular la transformada de Fourier por vectores de la imagen. Si bien, antes de esto se ha de trasponer la imagen, puesto que Matlab se trabaja con vectores columna, y OpenCV trabaja con vectores fila.

Lo siguiente es generar la máscara con la que se va a filtrar. Puesto que el filtro utilizado en el algoritmo es un filtro paso banda ideal, los píxeles de esta máscara tendrán valores 1

o 0 dependiendo de si la frecuencia que representan es admisible o no. Para generar esta máscara se ha creado un método que genera una imagen del tamaño del *slice* que está siendo procesado y en el cual se asignan los valores a los píxeles siguiendo la condición:

$$H_{IL}(u, v) = \begin{cases} 1 & \text{if } D(u, v) \leq D_0 \\ 0 & \text{if } D(u, v) > D_0 \end{cases}$$

Con , D la distancia en el espacio de frecuencias al origen de la imagen, es decir:

$$D = \frac{x}{w} * t;$$

Siendo *x* el píxel actual, *w* el ancho en píxeles del *slice*, y *t* el largo en píxeles del *slice*.

Por último, se multiplica elemento a elemento esta máscara con el *slice* temporal en espacio de frecuencias, con lo que filtramos las frecuencias que nos interesan (1 en la máscara).

El proceso es representado en la siguiente imagen:

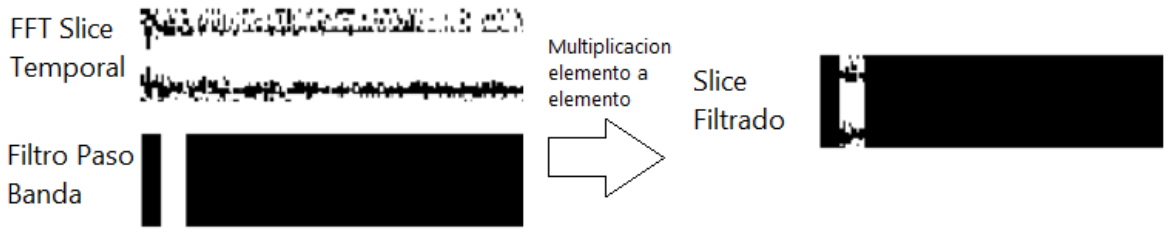

Figura 4-3 Representación gráfica del filtrado paso banda sobre un *slice* temporal en espacio de frecuencias.

Por último, se realiza la transformada de Fourier inversa sobre esta imagen y se traspone de nuevo.

Tras realizar el proceso para cada canal de color, se procede a juntar de nuevo todos los canales en una imagen RGB, y se realiza de nuevo un ShiftDim3C para volver a tener un vector de imágenes en las dimensiones Alto x Ancho x Tiempo.

4.1.1.4 Amplificación y reconstrucción

El primer paso es obtener las medias de color para los fotogramas filtrados, y las multiplicamos por un factor de amplificación, y le sumamos las medias de color de los fotogramas originales correspondientes. De esta forma se evita realizar una redimensión, se ahorran *(Alto x Ancho) - 1* multiplicaciones por canal de color (una por píxel), y se ahorra espacio en memoria respecto al método del algoritmo original.

Para calcular la media de cada canal se utiliza la función de OpenCV *mean*, la cual devuelve un vector con las medias de cada canal. Por este motivo, se ha creado un método para obtener tres vectores de medias a partir de un vector de vectores de medias.

4.1.1.5 Obtener frecuencia cardiaca

El algoritmo a seguir es el previamente descrito en la sección de diseño:

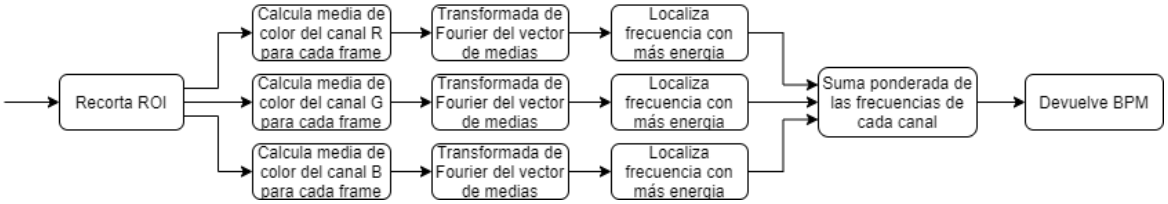

Figura 4-4 Diagrama de flujo para la estimación de la frecuencia cardiaca

La obtención de las medias se ha realizado en el paso anterior, pero se mantienen en un vector con tres dimensiones (una por canal de color) para poder realizar el resto del algoritmo como en el diseño original.

Para poder realizar las transformadas de Fourier de cada vector de medias sin depender de una librería adicional, en primer lugar, se han convertido los vectores desde el tipo de dato *"vector de float"* al tipo de dato matriz de OpenCV, convirtiendo cada vector en una matriz unidimensional.

Posteriormente se le ha aplicado una transformada de Fourier, utilizando OpenCV, y se han obtenido las magnitudes de los números complejos obtenidos al realizar la transformada.

El siguiente paso ha sido utilizar la función de OpenCV *minMaxLoc* que devuelve tanto los valores máximos y mínimos de una matriz, como la posición de ambos. Utilizando el índice del máximo, se obtiene la frecuencia que representa, utilizando la ecuación:

$$freq = \frac{fps}{2} * \frac{indice_{max}}{longitud_{vector} * 0.5}$$

Multiplicando esta frecuencia por 60, para convertirlo en pulsaciones por minuto, obtenemos la frecuencia cardiaca para el canal que se esté evaluando.

Posteriormente, se estima la frecuencia cardiaca realizando una media ponderada, en este caso utilizando el siguiente baremo [9]:

$$freq_{cardiaca} = freq_R * 0.299 + freq_G * 0.587 + freq_B * 0.114$$

4.1.2 Algoritmo de Luminancia

Para implementar este algoritmo se han tenido que realizar algunos cambios respecto al algoritmo de color, pero por lo general han sido poco significativos. A continuación, se detallan los cambios realizados en las etapas que los han requerido.

4.1.2.1 Generar pirámide Gaussiana

En esta etapa el cambio principal ha sido cambiar el espacio de color del fotograma antes de realizar la pirámide gaussiana. Para esto se ha utilizado la función de OpenCV *CvtColor*, con la propiedad *COLOR_BGR2YCrCB.*

Una vez convertido al espacio de color *YCrCB*, se realiza la pirámide gaussiana únicamente sobre el canal *Y*, obteniendo el nivel más alto de la pirámide, que es almacenado en un vector.

4.1.2.2 Filtrado Temporal

En esta etapa el cambio principal ha sido tener que crear una versión modificada de la función *ShiftDim3C* para que soporte imágenes de un canal, y en lugar de copiar valores de cada pixel como *Point3f* (puntos en el espacio de 3 dimensiones), copie un único valor *float*.

4.1.2.3 Amplificación y reconstrucción

El funcionamiento de este apartado es idéntico al del algoritmo de color, pero trabajando sobre un único canal, por lo que el vector de medias solo tiene una dimensión. En la sección 5.2.2.2 se han realizado pruebas y se ha comprobado que esta etapa se puede omitir sin afectar al funcionamiento del algoritmo (solo para el algoritmo de Luminancia).

4.1.2.4 Obtener frecuencia cardiaca

Para hallar la frecuencia cardiaca se repite el mismo proceso que en el algoritmo de color, pero trabajando únicamente sobre un canal, por lo que no resulta necesario realizar ninguna suma ponderada al final de la estimación, y se reducen la cantidad de transformadas de Fourier a realizar de tres a una.

A la hora de implementar este algoritmo y probar su rendimiento, se ha comprobado que la medición resulta idéntica si obtenemos la frecuencia cardiaca directamente de la salida de la operación del filtrado temporal, sin amplificar y reconstruir, lo que supone un ahorro en consumo memoria y en tiempo de ejecución.

Este no ha sido el caso para el algoritmo de color, en el cual reconstruir y amplificar la imagen proporciona resultados más precisos que si intentamos obtener la frecuencia cardiaca a partir de las imágenes procesadas sin reconstruir.

4.1.3 Mejoras algoritmo PPG

En esta sección se describe la implementación de las mejoras diseñadas para el algoritmo PPG, tanto a nivel de rendimiento como a nivel de precisión.

Los cambios realizados para mejorar el rendimiento han seguido el esquema descrito en la figura 3-12, siendo el cambio más significativo eliminar la necesidad de calcular repetidamente las medias de las caras almacenadas en el algoritmo, y realizar una única estimación tras obtener todas las medias de los fotogramas, y no una estimación por media.
Para mejorar la precisión del algoritmo PPG , se ha revisado el trabajo en el que está basado[15] para intentar añadir mejoras de precisión, y se ha añadido una interpolación linear en los valores de la media para que en lugar de recortar valores hasta el múltiplo de dos más cercano por abajo, se interpolen al múltiplo de dos más cercano por arriba. Esto no solo evita que se pierdan datos a la hora de procesar, sino que también habilita que el suavizado realizado por la ventana de *Hamming* funcione mejor por ajustarse al tamaño de ventana, ya que previamente al descartar valores, se perdía también el suavizado aplicado a los mismos. Por último, se ha modificado también el algoritmo para que soporte tamaños de ventana mayores a 256 fotogramas.

4.2 Ampliación y mejora de la aplicación

Una vez migrados los algoritmos a código en C++ usando OpenCV, se ha procedido a realizar los cambios necesarios en la aplicación para implementar el módulo en tiempo real y para facilitar la integración de nuevos algoritmos en la misma.

4.2.1 Módulo de Tiempo Real

En primer lugar, se diseñó la interfaz gráfica de usuario mediante la herramienta nativa de Visual Studio, para tener una primera versión de cuál sería el aspecto de la aplicación terminada. Posteriormente se desarrolló el módulo de la *webcam*, encargado de recibir imágenes desde la *webcam*, mostrarlas por pantalla y ejecutar algoritmos. Por último, se conectaron interfaz y módulo de la *webcam* para que fuese funcional.

4.2.1.1 Interfaz del módulo en tiempo real.

La primera versión de la interfaz estaba compuesta por un panel para mostrar los fotogramas que estaban siendo capturados, un botón para conectar con el sensor de pulsaciones, dos desplegables para seleccionar método de entrada y algoritmo, un bloque de texto para mostrar las pulsaciones que estaban registrándose con la pulsera, y otro bloque de texto para mostrar la estimación del algoritmo, un botón para iniciar el funcionamiento de la aplicación, y otro para pararlo.

Figura 4-5 Primera versión de la interfaz del módulo en tiempo real.

Posteriormente se decidió añadir una gráfica para mostrar la variación de las pulsaciones a lo largo del tiempo, tanto para la pulsera como para el algoritmo. En este punto se añadió la mejora mencionada en la sección 3.4, una gráfica a la pestaña de probar algoritmos para poder mostrar información relevante sobre la ejecución de estos, como las medias de color de los fotogramas de un video, o la transformada de Fourier de dichas medias.

En la sección 5.1 Se muestra el resultado final de la interfaz de la aplicación tras una ejecución en el modo *Directo* (Figura 5-2), y tras una ejecución desde el modo *Fichero* (Figura 5-1).

4.2.1.2 Módulo Webcam

Este módulo está diseñado para ejecutarse de manera paralela al hilo de la interfaz, por lo que se debe de crear un hilo antes de crear este objeto, y ejecutar su bucle principal desde dicho hilo.

Para acceder a la *webcam* del ordenador, y obtener imágenes, se ha utilizado *EmguCV*, un *wrapper* de OpenCV a lenguaje C#. Los componentes principales de la clase *WebcamModule* son:

- Una propiedad *VideoCapture* que almacenará la *webcam* como objeto accesible.
- Un flag para indicar si se ha detectado cámara.
- El clasificador que se utilizará para detectar caras en los fotogramas, por defecto un clasificador Haar.

Al crear el objeto del tipo módulo *webcam*, se carga el clasificador en cascada desde el disco. Posteriormente se intenta inicializar la *webcam*, y en caso de inicializarse correctamente, se ajusta la tasa de fotogramas a 30, y la exposición de la misma para prevenir que la imagen esté demasiado blanca ("quemada") lo cual puede generar problemas a la hora de recuperar información de la misma.

En caso de no poder inicializar la *webcam*, se cierra el hilo en el que se está ejecutando el módulo, y se muestra un mensaje de error en la interfaz.

El método *VideoFeed* se encarga de ejecutar el módulo, siguiendo el diagrama descrito en el capítulo 3.2. Esta función recibe como argumentos una *ImageBox*, que es el contenedor en el cual se van a mostrar los fotogramas; una estructura que contiene identificadores para otros elementos de la interfaz, como los bloques de texto o la gráfica; y el algoritmo que se va a utilizar para realizar las estimaciones.

En cada fotograma, el módulo ejecuta el clasificador en busca de caras. En caso de no encontrar ninguna, simplemente espera al siguiente fotograma. Se ha comprobado que el algoritmo tenía demasiado ruido en sus mediciones si se recortaba la cara detectada en cada fotograma, debido a que los clasificadores en cascada no siempre devuelven una región idéntica incluso para objetos estáticos. Por este motivo se ha decidido recortar la región detectada en el primer fotograma en los fotogramas consecutivos, siempre que en estos se detecte una cara.

Tras ello, se añaden los fotogramas al algoritmo, para que sean procesados durante la próxima ejecución del mismo. Para ello se llama al método *AddFrame* del algoritmo, que recibe como parámetro el recorte del fotograma que contiene la cara.

Para ejecutar el algoritmo desde el módulo de la *webcam*, se ha establecido por defecto una frecuencia de una ejecución por segundo, es decir, cada vez que se registran tantos fotogramas como la tasa de fotogramas a la que se está capturando. Para ejecutar el algoritmo, simplemente se crea un nuevo hilo que llama al método Run del algoritmo:

```
algorithmRun = new Thread(() => alg.Run());
algorithmRun.Start();
```

Figura 4-5 Código para ejecutar el algoritmo en paralelo

Cuando se ejecuta el algoritmo, se establece como verdadero el flag *waitingForEstimation*, que se utiliza para saber si se ha de comprobar el algoritmo en busca de una nueva estimación o no.

Para poder actualizar los elementos de la interfaz de una manera *thread-safe*, se ha hecho uso del *Dispatcher* de la interfaz, y una combinación de técnicas de programación funcional (funciones lambda y delegados) para realizar las actualizaciones necesarias. Por ejemplo, para obtener la lectura del sensor de pulsaciones desde el hilo de la *webcam*, que es el hilo encargado de actualizar la gráfica, se realiza la siguiente llamada al *Dispatcher*:

```
sensor.Dispatcher.Invoke(() => Double.TryParse(sensor.Text.Split(' ')[0], out sensorBPM));
```

Figura 4-6 Código para obtener la lectura del sensor de manera *thread safe*, siempre que este tenga un valor válido.

Se ha decidido actualizar la gráfica desde el propio hilo de la *webcam* puesto que las actualizaciones de la misma se realizan al obtener una estimación tras una ejecución del algoritmo seleccionado.

Por último, antes de pasar a la siguiente iteración del bucle, y tratar el siguiente fotograma, se actualiza el contenedor que muestra los fotogramas con el fotograma procesado, en el cual se dibuja un recuadro alrededor de la cara (si se ha detectado alguna).

En caso de que el flag *waitingForEstimation* esté establecido como verdadero, se realiza además una última comprobación, pues se revisa el *flag* llamado *LectureReady* del propio algoritmo para conocer si ha terminado de ejecutarse y tiene una estimación disponible, la cual se obtiene accediendo al atributo del algoritmo *LastEstimation*. Una vez obtenida la estimación, se baja el *flag waitingForEstimation*, se actualizan la gráfica y el bloque de texto que muestra la estimación en la interfaz, y se procede a procesar el siguiente fotograma.

4.2.2 Cambios en el módulo de Algoritmos

Para poder integrar los algoritmos en la aplicación, se ha reconsiderado el diseño del módulo de algoritmos para que resulte más sencillo añadir nuevos algoritmos, y permitir que se puedan ejecutar de forma más opaca, y con la menor dependencia externa posible.

Posteriormente, se ha procedido a implementar y adaptar los algoritmos para que fuesen compatibles con este nuevo diseño. También se han realizado cambios en algoritmo *PPG* para mejorar su funcionamiento y estabilidad.

4.2.2.1 Implementación de la clase abstracta Algoritmo

En primer lugar, se ha creado la clase abstracta *Algoritmo.cs* , de la que heredarán todos los nuevos algoritmos y que contendrá las cabeceras de las funciones básicas para el funcionamiento del algoritmo, y las propiedades que permitirán ajustar el tamaño de la ventana que puede procesar cada algoritmo, así como las frecuencias en las que se centrará.

Todos los algoritmos van a tener únicamente dos métodos, el método *Run*, que es un método abstracto que se deberá implementar en cada algoritmo en función del funcionamiento del mismo, y el método virtual *AddFrame*, que tiene una implementación por defecto de la que puede hacer uso un algoritmo que herede sin necesidad de

modificarse, pero que soporta ser modificado en caso de ser necesario procesar las caras antes de almacenarlas, o si se prefiere cambiar la ubicación en la que se almacenan.

La función *AddFrame* se ha decidido implementar haciendo uso de dos listas. Una de las listas es la que tendrá los fotogramas que ha de procesar el algoritmo, y la otra se utilizará para almacenar de manera temporal los fotogramas que se intenten añadir mientras el algoritmo está procesando los que tiene, para evitar modificar una lista mientras puede estar siendo accedida desde otro hilo. Para esto se ha utilizado un *flag* booleano llamado *Processing,* que se utiliza a modo de semáforo. Antes de añadir un fotograma a la lista de pendientes se comprueba si en la lista auxiliar hay fotogramas que estuviesen esperando para ser añadidos.

En este paso se ha implementado también la función *RunFromFile*, que lee los fotogramas desde una ruta en disco, y los añade al algoritmo para ser procesados mediante *AddFrame*.

En la propia clase algoritmo se han dejado preparadas una serie de listas, accesibles mediante métodos *get*, y pensadas para facilitar la depuración de los nuevos algoritmos que se puedan implementar. Para crear un nuevo algoritmo es suficiente con crear una nueva clase que herede algoritmo e implemente el método abstracto *Run* con el código para el algoritmo.

Ha sido necesario también modificar la clase *PPG.cs* ya existente en la aplicación, para que herede de la clase *Algoritmo.* A la vez que se realizaban estos cambios se han implementado las mejoras descritas en la sección 3.5.2.

4.3 Integración de los Algoritmos en la aplicación

La primera versión en OpenCV de los algoritmos se desarrolló en C++, puesto que es el lenguaje en el que está programado OpenCV, y por tanto las llamadas a las funciones se realizan de forma nativa.

Para poder llamar a los algoritmos desde la aplicación en C# se estudiaron dos posibilidades, una de ellas era crear un *wrapper* en C++/CLI para llamar a las funciones desde C# pero compiladas en C++, y la otra era migrar los algoritmos desde C++ a C# utilizando *EmguCV*. Se hizo una lista de pros y contras de ambos:

	Wrapper C++/CLI	*EmguCV*
Pros	+En teoría mejor rendimiento, pues solo necesita un vector de fotogramas	+Más mantenible +No requiere más dependencias externas +Todo el código para un algoritmo en la misma clase
Contras	-Menos mantenible -Añade dependencias externas adicionales -Mayor complejidad en el propio código	- Hay que hacer algunas llamadas adicionales para convertir tipos de datos antes de procesarlos

Tabla 4-2 Pros y contras de ambos métodos para llamar a los algoritmos desde C#

En principio se comenzó con el desarrollo del *wrapper*, puesto que pese a ser menos mantenible y más complejo, el rendimiento debería ser mejor y el objetivo es poder usarlos en tiempo real, pero para poder pasar el tipo de datos de las matrices en OpenCV, *Mat*,

desde C++ a C#, el tipo de datos debe ser "*Managed*", pero esto no es así, por lo que sería necesario especificar como se gestiona, no siendo esto una labor sencilla, y además podría presentar problemas de compatibilidad con el tipo de datos *Mat* de *EmguCV*, el *wrapper* de OpenCV que se está utilizando en la aplicación de C#.

Por este motivo, se ha decidido reescribir en primera instancia los algoritmos en C# haciendo uso de *EmguCV*, y en caso de que el rendimiento hubiese sido suficientemente inferior al ofrecido al ejecutarse desde C++, se retomaría la alternativa del *wrapper*.

Por suerte, el rendimiento de los algoritmos es más que suficiente para ejecutarse en tiempo real, por lo que ha bastado con implementarlos desde C# utilizando *EmguCV*.

4.3.1 Integración del Algoritmo de Color

Puesto que al final se ha optado por hacer todas las llamadas a OpenCV desde la librería *EmguCV*, se ha tenido que pasar el código del algoritmo desde C++ a C#. Esta traducción ha sido bastante inmediata puesto todas las funciones de OpenCV utilizadas en el algoritmo en C++, se encuentran en *EmguCV*.

Para el algoritmo de color se ha creado la clase *ColorAlg.cs*, y se han creado las funciones necesarias para la ejecución del algoritmo imitando el código desarrollado en C++, con los cambios necesarios en parámetros de funciones, nombres de las mismas y el almacenamiento de los resultados, puesto que en C++ se realizaban mediante pasos por referencia en los argumentos, y en C# se obtienen como *return* de una función.

En principio se optó por mantener el método *AddFrame* por defecto, y cada vez que se ejecute el algoritmo procesar los fotogramas, pero tras implementarlo se comprobó que se podía realizar el procesamiento básico de la imagen (generar la pirámide gaussiana) en el instante en el que se añadía el fotograma a la imagen, sin perder rendimiento en tiempo real, puesto que el tiempo de ejecución medio de este proceso son alrededor de 3 ms, y a cambio al ejecutar el algoritmo le liberamos de 3 ms por cada fotograma que tenga que procesar, lo que lo hace más responsivo, principalmente con mayores tamaños de ventana.

4.3.2 Integración del Algoritmo de Luminancia

Del mismo modo que para el algoritmo de color, se ha vuelto a programar el algoritmo en C#, añadiendo los cambios en el espacio de color, y cambiando las funciones para trabajar con un único canal en lugar de 3.

La clase que se ha creado para implementar este algoritmo ha sido *LuminanciaAlg.cs*. De nuevo, solo ha sido necesario cambiar los nombres de algunas funciones, sus parámetros y sus retornos respecto al código de C++ para este algoritmo.

En este caso en el método *AddFrame*, además de realizar la pirámide gaussiana se ha incluido el cambio del espacio de color desde RGB a YCrCb.

5 Pruebas y resultados

Una vez terminado el proceso del desarrollo de la aplicación y algoritmos, se ha procedido a realizar pruebas para evaluar su funcionamiento y precisión con videos en tiempo real, así como para encontrar posibles puntos débiles de los algoritmos, y preparar el trabajo futuro a realizar sobre los mismos.

5.1 Estado final de la aplicación

La aplicación, en su estado final, permite utilizar los nuevos algoritmos tanto desde el modo de probar algoritmos como desde el modo en tiempo real.

En el modo *Fichero* ahora se muestra además una gráfica, que en la Figura 5-1 representa la FFT de la cual se extrae la frecuencia cardiaca, pero que también puede utilizarse para mostrar la media de color y la frecuencia cardiaca.

Figura 5-1 Modo *Fichero*, prueba algoritmo Luminancia

En el desplegable con la etiqueta “Algoritmo” se puede elegir el algoritmo a utilizar, entre Color, Luminancia y PPG, este último con las mejoras implementadas en la sección 4.2.2.2.

El aspecto de la interfaz tras incorporar el módulo de análisis en tiempo real tiene el siguiente aspecto, tras ejecutarse para el algoritmo PPG (descrito más en profundidad en el apartado 4.3), durante un total de 50 segundos:

Figura 5-2 Captura de una ejecución del modo Tiempo Real con el algoritmo PPG

Al arrastrar el ratón por encima de la gráfica se obtiene información sobre que valores representa cada serie de puntos:

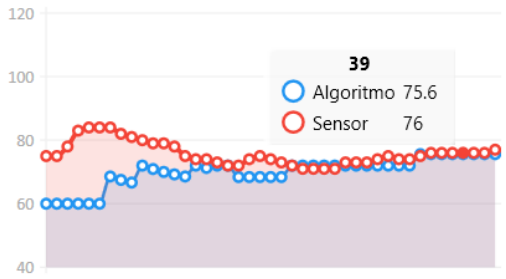

Figura 5-3 Gráfica con información de las series que representa

5.2 Pruebas de los algoritmos sobre videos

Se ha evaluado el correcto funcionamiento de los algoritmos desde el modo *Fichero* de la aplicación, utilizando las grabaciones ya existentes en la propia aplicación, la cual cuenta con una base de datos "*Nueva"* que contiene los videos grabados por Julia, y una base de datos "*Antigua"* con los videos grabados por Ana. Se han añadido además a la aplicación dos vídeos del MIT [10] , de mayor duración a los ya incorporados en la aplicación, para contrastar el rendimiento de los algoritmos con videos externos. Para estos videos se tiene un valor de *Ground Truth* de 54 y 55 pulsaciones por minuto.

En la siguiente sección se especifican las pruebas realizadas y los resultados obtenidos para cada algoritmo al evaluarlos sobre vídeos previamente grabados.

5.2.1 Pruebas Algoritmo de Color

En primer lugar, se han realizado pruebas para el algoritmo de color desde el modo *Fichero*. Los mejores resultados se han obtenido utilizando los videos del MIT, y algunos de los videos de la base de datos *Nueva*. También se ha implementado una variación en la forma de detectar la cara en el algoritmo (ver sección 3.2.1.1), y se han analizado los resultados de esta variación.

5.2.1.1 Pruebas en modo Algoritmos

Para el vídeo de la base de datos *Nueva* con título *17-01-2019_18-18-29*, y ambos videos del MIT se han obtenido unas estimaciones bastante acertadas, y en el caso del video del MIT a 54 pulsaciones, la diferencia con las pulsaciones reales se ha comprobado que se debe a resolución del espacio de frecuencias. En la figura 5-4 se puede observar el resultado de la ejecución para estos tres videos.

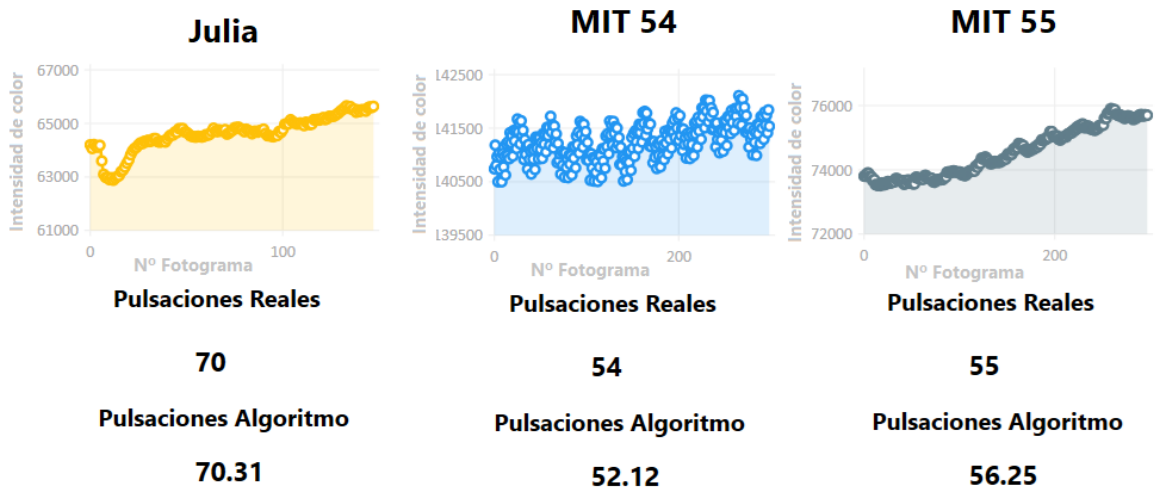

Figura 5-4 Estimaciones para tres videos utilizando el algoritmo de color.

Al probar este algoritmo sobre otros videos en los cuales el sujeto se mueve ligeramente, se obtienen unos resultados con mucho ruido, de los cuales no es posible extraer la frecuencia cardiaca correctamente. Este ruido resulta más evidente al comparar las transformadas de Fourier de las medias, puesto que se puede comprobar que no solo hay más picos, sino que la energía está más distribuida, es decir, está detectando más frecuencias además de la cardiaca, con una intensidad similar.

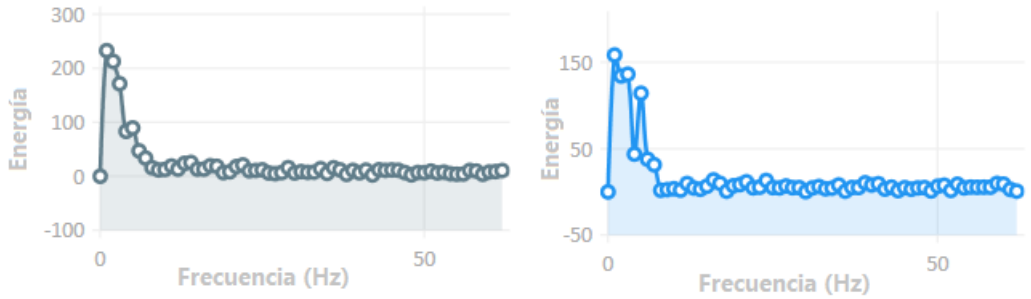

Figura 5-5 Comparativa Estimación Correcta (Izquierda) y estimación incorrecta (derecha)

5.2.1.2 Cambio en la detección de la cara

En la sección de diseño se ha mencionado la posibilidad de intentar mejorar la precisión detectando la cara en cada fotograma en lugar de una detección y recortar la misma sección para todos los videos. Se ha probado esta aproximación para comprobar el rendimiento de la misma, y se ha podido observar que añade una gran cantidad de ruido a la medición.

En la figura 5-3 se muestra un contraste de los valores para la media de color de cada fotograma en el video del MIT a 54 pulsaciones cuando se recorta la misma región en todos los fotogramas y cuando se detecta la cara en cada fotograma:

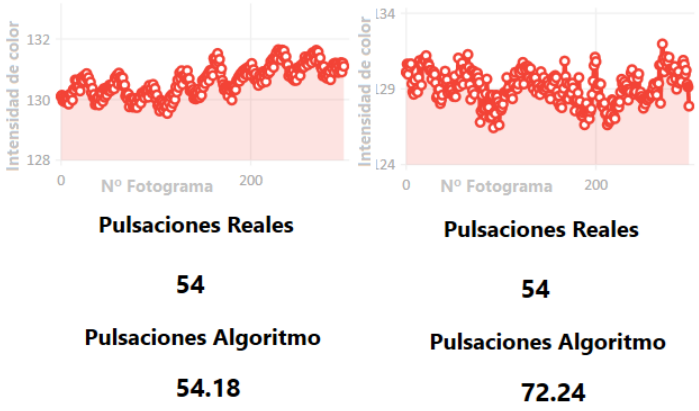

Figura 5-6 Comparativa medias de color de los fotogramas al utilizar detección única de la cara y recorte (izquierda) y detección constante de la cara (derecha)

Otro de los problemas que se presentan al realizar una detección del rostro en cada fotograma es que el tiempo de ejecución aumenta considerablemente, pasando de aproximadamente 600ms a alrededor de 9 segundos.

Como este cambio supone pérdida de precisión y rendimiento, se ha descartado y se ha decidido mantener el modelo inicial de detección del rostro en el primer fotograma y recorte de esa región en todos.

5.2.2 Pruebas algoritmo Luminancia

Para el algoritmo luminancia, además de realizar pruebas para comprobar su correcto funcionamiento, se ha contrastado el funcionamiento del mismo realizando la amplificación y reconstrucción, y estimando la frecuencia directamente sobre la señal filtrada temporalmente.

5.2.2.1 Pruebas desde el modo Algoritmos

Las pruebas se han realizado sobre los videos considerados en el apartado anterior, y de nuevo los resultados han sido similares.

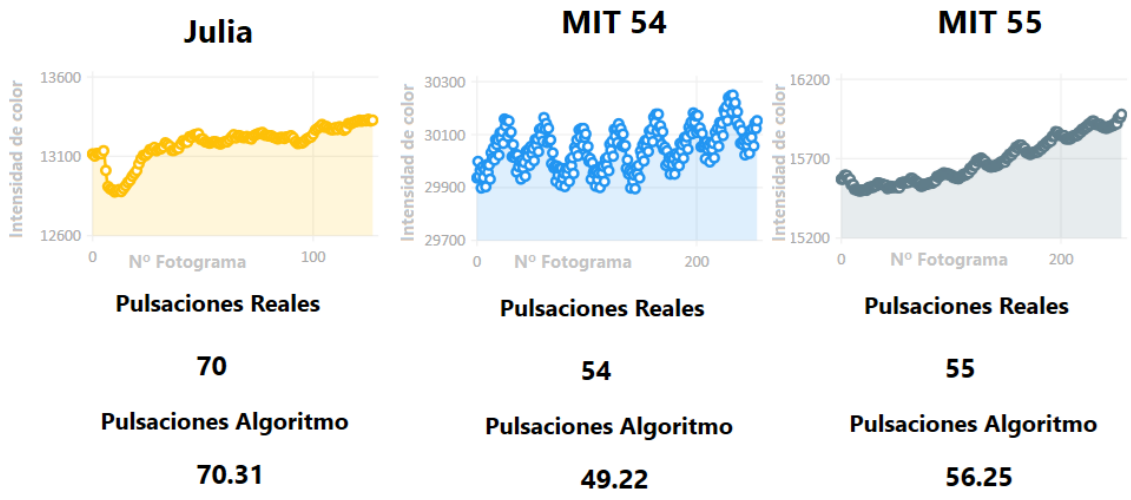

Figura 5-7 Estimaciones para tres videos utilizando el algoritmo de luminancia

Se puede observar una peor estimación para el video del MIT 54. Esta estimación se ha intentado mejorar analizando únicamente la variación de la luminancia en la región de la frente, pero el resultado obtenido ha sido el mismo. Al analizar el resultado únicamente en la región de la frente se ha obtenido el mismo resultado, tanto recortando antes de filtrar temporalmente, como recortando sobre la señal filtrada.

5.2.2.2 Amplificar y Reconstruir

Se han realizado pruebas para contrastar el funcionamiento del algoritmo si se amplificaba y reconstruía la señal antes de realizar la estimación, así como si el rendimiento variaba al analizar únicamente la región de la frente tras amplificar y reconstruir. En la figura 5-7 se muestran las medias de color para los vídeos MIT 55 pulsaciones y *17-01-2019_18-18-29* tras realizar la estimación sobre la señal filtrada, tras realizarla sobre la señal reconstruida y tras estimar únicamente sobre la región de la frente de la señal reconstruida.

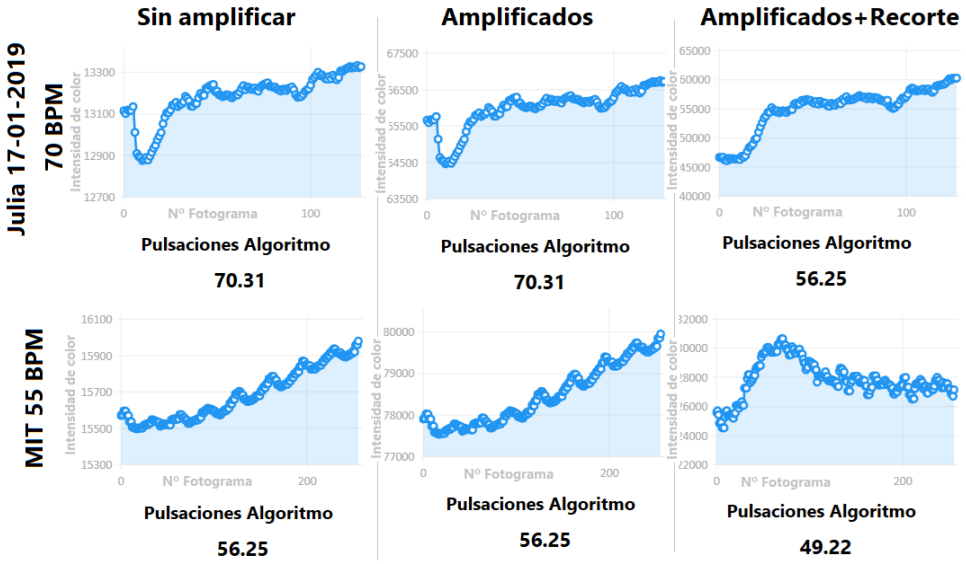

Figura 5-8 Comparativa de resultados para las mediciones sobre dos videos con 3 variaciones del algoritmo

Se ha comprobado que el método de amplificación y recorte introduce demasiado ruido y por tanto reduce la precisión del algoritmo. Se ha probado en más videos el algoritmo amplificado y sin amplificar, y se ha llegado a la conclusión de que no se ve afectado por el factor de amplificación, salvo que este sea muy pequeño, puesto que en ese caso se realiza la medición sobre el vídeo original.

Que no haya diferencia en las mediciones, junto al hecho de que omitir la fase de amplificación y reconstrucción aumenta el rendimiento del algoritmo y reduce la cantidad de memoria utilizada, ha llevado a aplicar la modificación de omitir el paso de amplificar y reconstruir del algoritmo.

5.2.3 Pruebas PPG mejorado

Para realizar las pruebas del algoritmo PPG mejorado, se han contrastado los resultados con los obtenidos de la aplicación en su estado original, a la cual se le han añadido también los videos del MIT, puesto que son sujetos estáticos, y son videos de mayor duración que los existentes en la aplicación.

Los resultados obtenidos han sido los siguientes:

Video	*Duración (s)*	*Valor Real (bpm)*	*PPG (bpm)*	*PPG Mejorado (bpm)*	*Error PPG (%)*	*Error PPG Mejorado (%)*	*∆ Error (%)*
MIT 54 BPM	10	54.00	55.66	54.18	3.07	0.03	- 3.04
MIT 55 BPM	10	55.00	79.93	54.36	45.32	0.64	- 44.68
Julia-1	5	69.11	65.32	60.81	5.48	12.00	+ 6.52
Julia-2	5	55.11	65.44	55.38	18.07	0.48	- 17.59

Tabla 5-1 Comparativa de rendimiento entre el PPG original y el PPG mejorado

Julia 1 se refiere al video de la base de datos *Nueva* con nombre *17-01-2019*, y *Julia-2* se refiere al video *16-06-2019-23-01-12* de la misma base de datos.

Se ha podido comprobar que la precisión del algoritmo aumenta de manera considerable cuando aumenta el tamaño de ventana (la duración del vídeo), lo cual puede explicar por qué en algún caso se ha obtenido una estimación peor sobre los videos pregrabados.

Además, el tiempo de ejecución se ha reducido de manera considerable, en la versión mejorada del algoritmo PPG, pasando en el caso del video del MIT a 54 BPMs de 9.5 segundos en 10 ejecuciones a menos de 1 segundo por ejecución. En el caso del video *16-06-2019_23-01-12* se ha reducido el tiempo de ejecución de 14 segundos con el algoritmo original a tan sólo 2 segundos con la versión mejorada.

5.3 Prueba aplicación en tiempo real

Se han realizado diferentes pruebas para la aplicación en tiempo real, empezando por contrastar el rendimiento y precisión de los algoritmos ejecutados en tiempo real, para después ver cómo le afecta a este rendimiento y precisión cambios en las frecuencias filtradas y en el tamaño de ventana.

El tamaño inicial de ventana se ha ajustado a 300 fotogramas, y se ha configurado la aplicación para realizar una estimación cada 30 fotogramas (a 30 fotogramas por segundo).

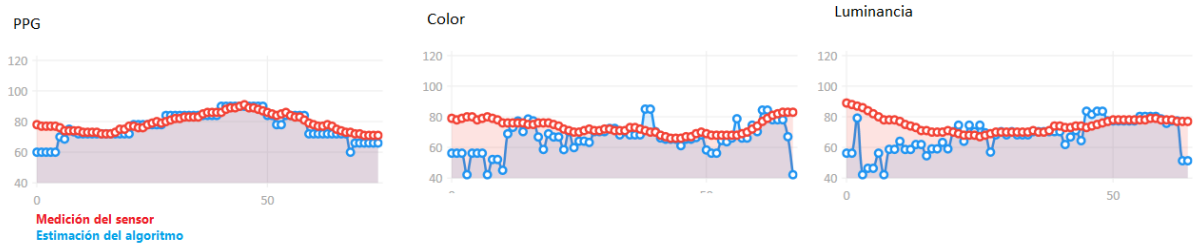

Figura 5-9 Gráficas para las estimaciones utilizando diferentes algoritmos

A partir de diferentes ejecuciones para este tamaño de ventana se han podido comprobar diferentes ventajas y desventajas en los algoritmos, las cuales se contrastan en la siguiente tabla:

Algoritmo	*Ventajas*	*Desventajas*
PPG	+Menos susceptible al ruido.	-Tarda un poco en reaccionar a los cambios de frecuencia
Color y Luminancia	+Mucho más susceptibles al ruido. +Más pesados computacionalmente.	-En condiciones óptimas estima la frecuencia cardiaca mucho más rápido que el algoritmo PPG

Tabla 5-2 Ventajas y desventajas de cada algoritmo integrado en la aplicación

Se ha comprobado que la susceptibilidad al ruido de los algoritmos de color y luminancia está producida en gran medida por el hecho de que se realizan dos filtrados en espacio de frecuencias. Al realizar varias pruebas se ha podido comprobar que cuanto mayor es la resolución de la cara, menor es el impacto del ruido en la medición (siempre que esta esté estática). También se ha comprobado que estos algoritmos funcionan mejor con tamaños de ventana no muy grandes. Para un tamaño de ventana de 500 fotogramas resulta prácticamente imposible estimar una frecuencia cardiaca de manera estable salvo que las condiciones de luminosidad sean idóneas.

Por otro lado, el algoritmo PPG se ha comprobado que resulta mucho más estable la medición cuanto mayor es el tamaño de ventana, pero también supone que tarda más en reaccionar a cambios en la frecuencia cardiaca (puesto que tarda más en cambiar la frecuencia en la que se acumula más energía).

6 Conclusiones y trabajo futuro

6.1 Conclusiones

El objetivo de este trabajo era mejorar y ampliar la aplicación ya existente en el VPULab para que fuese capaz de trabajar sobre secuencias de vídeo obtenidas en tiempo real, y además adaptar una serie de algoritmos ya existentes para que pudiesen ser ejecutados desde esta aplicación.

Se ha conseguido implementar estos objetivos cumpliendo los objetivos principales. El módulo en tiempo real funciona de manera satisfactoria, y el hecho de que se haya realizado para ejecutarse en paralelo permite que en un futuro se puedan realizar estimaciones utilizando algoritmos computacionalmente más pesados, sin afectar al rendimiento de la aplicación. Se ha conseguido migrar los algoritmos desde Matlab a OpenCV, y el hecho de tener los algoritmos separados en clases nos facilita su mantenimiento, modificación o incluso extensión utilizando herencia de clases.

Las mejoras de rendimiento en los algoritmos permiten además realizar las estimaciones de manera más rápida, y ocupando menos memoria.

6.2 Trabajo futuro

Durante el trabajo desarrollado se han encontrado posibles frentes en los que sería interesante avanzar para mejorar la precisión de los algoritmos ya implementados, así como ampliar aún más la funcionalidad de la aplicación:

- Conseguir un método para detección y seguimiento de un rostro que introduzca el menor ruido posible en la región seguida, para poder detectar las pulsaciones de una persona en movimiento.
- Añadir la posibilidad de realizar estimaciones en tiempo real utilizando otras cámaras (e.g., Kinect).
- Añadir algoritmos a la aplicación que sean capaces de estimar frecuencia cardiaca utilizando imágenes de profundidad (captadas mediante la Kinect -en la versión anterior el módulo de grabación ya contemplaba la grabación de *datasets* multimodales).
- Aumentar el soporte de otros modelos de cámaras para captación y grabación.
- Incluir el soporte de otros modelos de pulsímetros, tanto para el módulo de grabación como el de *Directo*.
- Modificar el módulo de grabación para que pueda grabar utilizando una *webcam* además de una Kinect.

Referencias

[1] C. Umeh, «Systole vs. diastole: The differences». https://nccmed.com/systole-vs-diastole-the-differences/. (accedido mayo 2020)

[2] S. El-Samad, O. Obeid, G. Zaharia, S. Sadek, y G. El Zein, «Remote Heartbeat Detection Using Microwave System from Four Positions of a Normally Breathing Patient», *IRECAP*, vol. 6, n.º 3, p. 175, jun. 2016, doi: 10.15866/irecap.v6i3.9281.

[3] Y. Gu, X. Zhang, Z. Liu, y F. Ren, «WiFi-Based Real-Time Breathing and Heart Rate Monitoring during Sleep», en *Proc. 2019 IEEE Global Communications Conference (GLOBECOM)*, Waikoloa, HI, USA, dic. 2019, pp. 1-6, doi: 10.1109/GLOBECOM38437.2019.9014297.

[4] L. Scalise, U. Morbiducci, y M. De Melis, «A laser Doppler approach to cardiac motion monitoring: effects of surface and measurement position», en *Proc. Seventh International Conference on Vibration Measurements by Laser Techniques: Advances and Applications*, Ancona, Italy, jun. 2006, p. 63450D, doi: 10.1117/12.693151.

[5] I. Pavlidis y J. Levine, «Thermal image analysis for polygraph testing», *IEEE Eng. Med. Biol. Mag.*, vol. 21, n.º 6, pp. 56-64, nov. 2002, doi: 10.1109/MEMB.2002.1175139.

[6] W. Zeng, Q. Zhang, Y. Zhou, G. Xu, y G. Liang, «Infrared video based non-invasive heart rate measurement», en *Proc. 2015 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, Zhuhai, dic. 2015, pp. 1041-1046, doi: 10.1109/ROBIO.2015.7418909.

[7] J. Allen, «Photoplethysmography and its application in clinical physiological measurement», *Physiol. Meas.*, vol. 28, n.º 3, pp. R1-R39, mar. 2007, doi: 10.1088/0967-3334/28/3/R01.

[8] W. Wang, A. C. den Brinker, S. Stuijk, y G. de Haan, «Algorithmic Principles of Remote PPG», *IEEE Trans. Biomed. Eng.*, vol. 64, n.º 7, pp. 1479-1491, jul. 2017, doi: 10.1109/TBME.2016.2609282.

[9] W. Verkruysse, L. O. Svaasand, y J. S. Nelson, «Remote plethysmographic imaging using ambient light», *Opt. Express*, vol. 16, n.º 26, p. 21434, dic. 2008, doi: 10.1364/OE.16.021434.

[10] H.-Y. Wu, M. Rubinstein, E. Shih, J. Guttag, F. Durand, y W. T. Freeman, «Eulerian Video Magnification for Revealing Subtle Changes in the World», *ACM Transactions on Graphics*, nº 65, p. 8., jul 2012

[11] A. M. Unakafov, «Pulse rate estimation using imaging photoplethysmography: generic framework and comparison of methods on a publicly available dataset», *Biomed. Phys. Eng. Express*, vol. 4, n.º 4, p. 045001, abr. 2018, doi: 10.1088/2057-1976/aabd09.

[12] G. de Haan y V. Jeanne, «Robust Pulse Rate From Chrominance-Based rPPG», *IEEE Trans. Biomed. Eng.*, vol. 60, n.º 10, pp. 2878-2886, oct. 2013, doi: 10.1109/TBME.2013.2266196.

[13] A. Martín Doncel, «Detección de ritmo cardíaco mediante análisis de secuencias de vídeo en color», Trabajo Fin de Grado, Grado en Ingeniería en Tecnologías y Servicios de Telecomunicación, Universidad Autónoma de Madrid, 2018.

[14] J. Simón Chico, «Aplicación de captura de datasets y demostración de algoritmos de detección de ritmo cardíaco mediante análisis de secuencias de vídeo», Trabajo Fin

de Grado, Grado en Ingeniería en Tecnologías y Servicios de Telecomunicación, Universidad Autónoma de Madrid, 2019

[15] M. Mozifian, «Affective Mirror: Automated emotion detection through photoplethysmography & facial expression analysis», *Bachelor's Thesis, University of St Andrews*, Abr 2014

[16] F. Molina Sanz, «Detección de ritmo cardíaco mediante análisis de secuencias de vídeo » Trabajo Fin de Grado, Grado en Ingeniería en Tecnologías y Servicios de Telecomunicación, Universidad Autónoma de Madrid, 2019.

[17] G. Balakrishnan, F. Durand, y J. Guttag, «Detecting Pulse from Head Motions in Video», en *Proc. 2013 IEEE Conference on Computer Vision and Pattern Recognition*, Portland, OR, USA, jun. 2013, pp. 3430-3437, doi: 10.1109/CVPR.2013.440.

[18] C. Tomasi y T. Kanade, «Detection and Tracking of Point Features», *Technical Report CMU-CS-91-132*, Abr 1991

[19] J.-P. Lomaliza y H. Park, «Improved Heart-Rate Measurement from Mobile Face Videos», *Electronics*, vol. 8, n.º 6, p. 663, jun. 2019, doi: 10.3390/electronics8060663.

[20] P. Viola y M. Jones, «Rapid object detection using a boosted cascade of simple features», en *Proc. 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001*, Kauai, HI, USA, 2001, vol. 1, pp. I-511-I-518, doi: 10.1109/CVPR.2001.990517.

[21] K. Kadir, M. K. Kamaruddin, H. Nasir, S. I. Safie, y Z. A. K. Bakti, «A comparative study between LBP and Haar-like features for Face Detection using OpenCV», en *Proc. 2014 4th International Conference on Engineering Technology and Technopreneuship (ICE2T)*, Kuala Lumpur, Malaysia, ago. 2014, pp. 335-339, doi: 10.1109/ICE2T.2014.7006273.

Glosario

ECG	Electrocardiograma
VCG	Vibrocardiograma
RGB	Red Green Blue (Espacio de color)
FFT	Fast Fourier Transform
DFT	Direct Fourier Transform
ICA	Independent Component Analysis
PCA	Principal Component Analysis
BSD	Berkeley Software Distribution
LBP	Local Binary Patterns
KLT	Lukas-Kanade Tracker
ROI	Region of Interest
PPG	PhotoPlethysmography
CLI	Common Language Interface
MIT	Massachusetts Institute of Technology

Anexos

A Manual de Usuario

A-1 Manual de Instalación

En primer lugar, se deben cumplir unos requisitos hardware para poder utilizar todas las nuevas funciones de la aplicación.

- Una cámara web, puede ser integrada en un portátil o USB.
- Un sensor de frecuencia cardiaca Polar H7
- Adaptador de *Bluetooth* en el ordenador en el que se ejecute la aplicación, si no lo tiene integrado.
- Una Kinect y un adaptador de Kinect para PC (para el módulo de grabación, puesto que no soporta grabación con *webcam*).

A nivel de software se requerirán:

- Un ordenador con Windows 10.
- .NET FRAMEWORK 4.6.1 instalado en el equipo.

Los módulos adicionales requeridos por la aplicación, *EmguCV* y *Live Charts*, se han instalado utilizando la herramienta *NuGet* de Visual Studio, y se encuentran en los archivos locales del proyecto, pero pueden descargarse desde el *NuGet Package Manager* en caso de requerirse.

También se ha utilizado la librería AForge para realizar las transformadas de Fourier, que también se encuentra en el proyecto, pero puede descargarse desde la página oficial www.aforgenet.com/framework/downloads.html en caso de ser necesario.

A-2 Manual de la aplicación

Al ejecutar la aplicación se empieza en el modo *Grabación.* Este modo no ha sido modificado y requiere una Kinect para ser utilizado. Para más detalles, consultar [14].

En la parte superior izquierda podemos cambiar de modo mediante un sistema de pestañas.

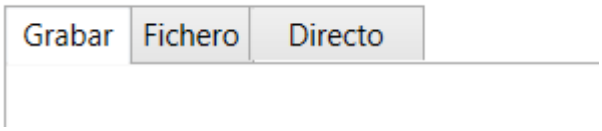

Figura A-1 Selección de modo de la aplicación mediante pestañas

Seleccionando el modo *Fichero*, nos aparece la siguiente ventana:

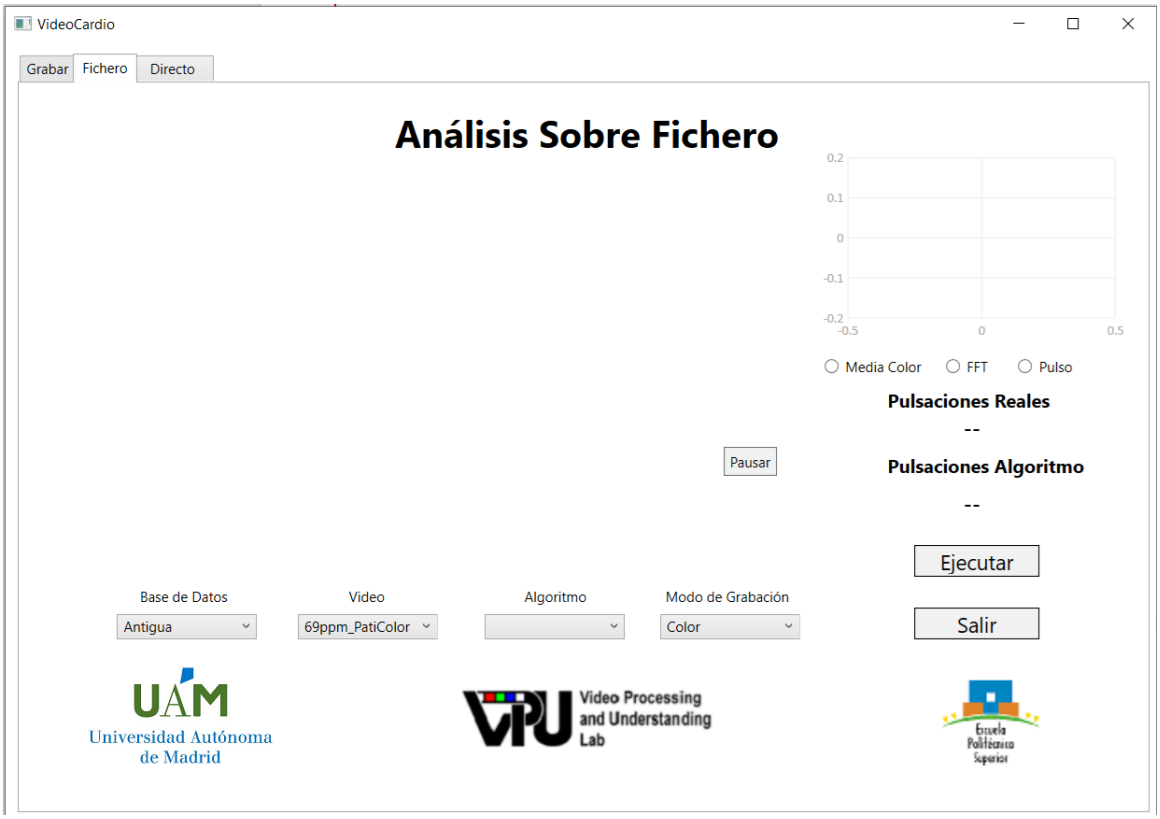

Figura A-2 Ventana del Modo *Fichero*

En esta ventana podemos elegir el video a analizar y el algoritmo a utilizar para estimar la frecuencia cardiaca, mediante los desplegables de la parte inferior de la pantalla. Además, en la parte superior derecha tenemos una gráfica en la cual podemos elegir que información del algoritmo mostrar.

Figura A-3 Modo *Fichero* tras una ejecución de un algoritmo

Al cambiar la información que queremos que se nos muestre, obtenemos diferentes gráficas:

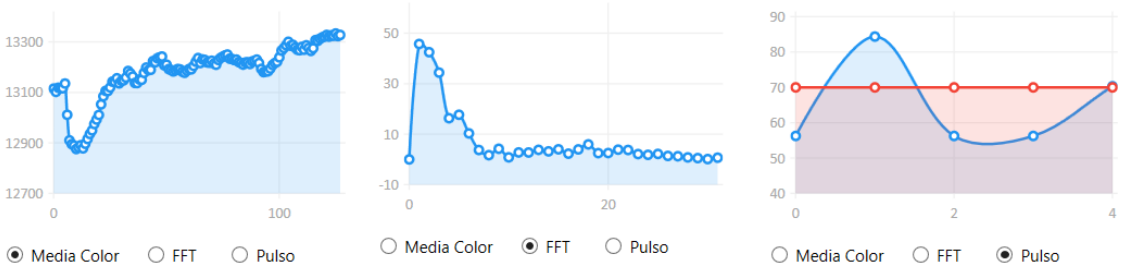

Figura A-4 Valores que puede mostrar la gráfica del modo *Fichero*

En la siguiente pestaña, *Directo*, podemos probar los algoritmos sobre videos en directo mediante una *webcam*. Para este paso será necesario disponer de una *webcam* y un pulsímetro, el cual no es necesario si únicamente se quiere estimar las pulsaciones y no comparar esta medición con las pulsaciones reales.

Para ejecutarlo, se debe primero seleccionar un algoritmo en el desplegable, y darle a empezar. Se debe permanecer lo más quieto posible dentro del recuadro, y con luz no variable para prevenir errores en la estimación. En la gráfica complementaria se muestra una comparativa entre los valores estimados y los registrados reales.

Figura A-5 Modo *Directo* tras una ejecución.

Se puede pulsar el botón de parar en cualquier instante para parar el módulo, pero se mantienen los resultados de la gráfica hasta la siguiente ejecución.

Si durante la ejecución se deja de detectar una cara durante un número determinado de fotogramas (modificable desde el código) se reinicia la estimación, y se vacía la gráfica.

B Manual del programador

A continuación, se añaden ciertas guías que pueden ser útiles para realizar modificaciones futuras sobre lo realizado en este Trabajo de Fin de Grado.

B.1 Funcionalidades y su ubicación en ficheros

A continuación, se muestran una lista de funcionalidades y las clases en las que se encuentra su código, para que en caso de querer hacer alguna modificación, se pueda localizar el código más rápidamente.

Funcionalidad	*Clase*	*Función*
Ejecutar algoritmo (*Fichero*) **	MainWindow.xaml.cs	Ejecutar_btn_Click
Mostrar video (*Fichero*) *	MainWindow.xaml.cs	Ejecutar_btn_Click
Insertar valores gráfica (*Fichero*)	MainWindow.xaml.cs	Ejecutar_btn_Click
Algoritmo de Color	ColorAlg.cs	Run
Algoritmo de Luminancia	LuminanciaAlg.cs	Run
Algoritmo PPG	PPG.cs	Run
Ejecutar algoritmo (*Directo)*	MainWindow.xaml.cs	Start_LF
Parar algoritmo (*Directo)*	MainWindow.xaml.cs	Parar_LF
Ejecutar módulo de la cámara	MainWindow.xaml.cs	StartLiveFeed
Bucle de la cámara	WebcamModule.cs	VideoFeed

Tabla B-1 Funcionalidades y archivos donde está el código correspondiente

Las funcionalidades marcadas con un * fueron desarrolladas por Julia Simón, y por tanto no se detallan en este trabajo, referirse a [14].

Las funcionalidades marcadas con ** fueron cambios menores sobre el código de Julia Simón, para añadir funcionalidad o adaptar cabeceras de funciones.

El diseño y la implementación del resto de funcionalidades se encuentra descrito en las secciones de diseño y desarrollo de este trabajo.

B.2 Añadir nuevos algoritmos

El procedimiento para añadir nuevos algoritmos es el siguiente:

1. Se crea una clase para el nuevo algoritmo, esta clase debe heredar de la clase abstracta *Algoritmo.cs*.
2. Se implementa el método *Run* que contendrá el código para el desarrollo del algoritmo. Se debe marcar el *flag LectureReady* para indicar que hay una estimación disponible tras ejecutarse, y almacenar la estimación en el atributo *LastEstimation.* Se puede utilizar el *flag Processing* para evitar múltiples ejecuciones concurrentes del algoritmo (usar como un semáforo).

3. Si es necesario, sobrecargar el método *AddFrame* para almacenar en otra lista los datos que se van a procesar cada vez que se ejecute el algoritmo, o para aplicar un preprocesamiento sobre los mismos.
4. Asegurarse de almacenar las listas pensadas para *debug*. Si se trabaja con algoritmos sobre medias de color, almacenar las mismas en el atributo lista *means* , y en caso de realizar transformadas de Fourier, almacenar los valores en cada punto de la misma en el atributo de la clase algoritmo *fft*.

Por último, para poder ejecutar el algoritmo, se deberán realizar los siguientes cambios:

1. Añadir en las listas de algoritmos de la interfaz (*ComboBox*) un elemento para identificar el nuevo algoritmo, tanto en la pestaña de *Fichero* como en la pestaña *Directo.*
2. Añadir en la función *ejecutar_btn_Click* del archivo *MainWindow.xaml.cs* una nueva condición que compruebe si el ítem seleccionado del *combobox* de algoritmos coincide con el nombre del algoritmo.
3. Añadir en la función *StartLiveFeed* archivo *MainWindow.xaml.cs* una nueva condición que compruebe si el ítem seleccionado del *combobox* de algoritmos coincide con el nombre del algoritmo.